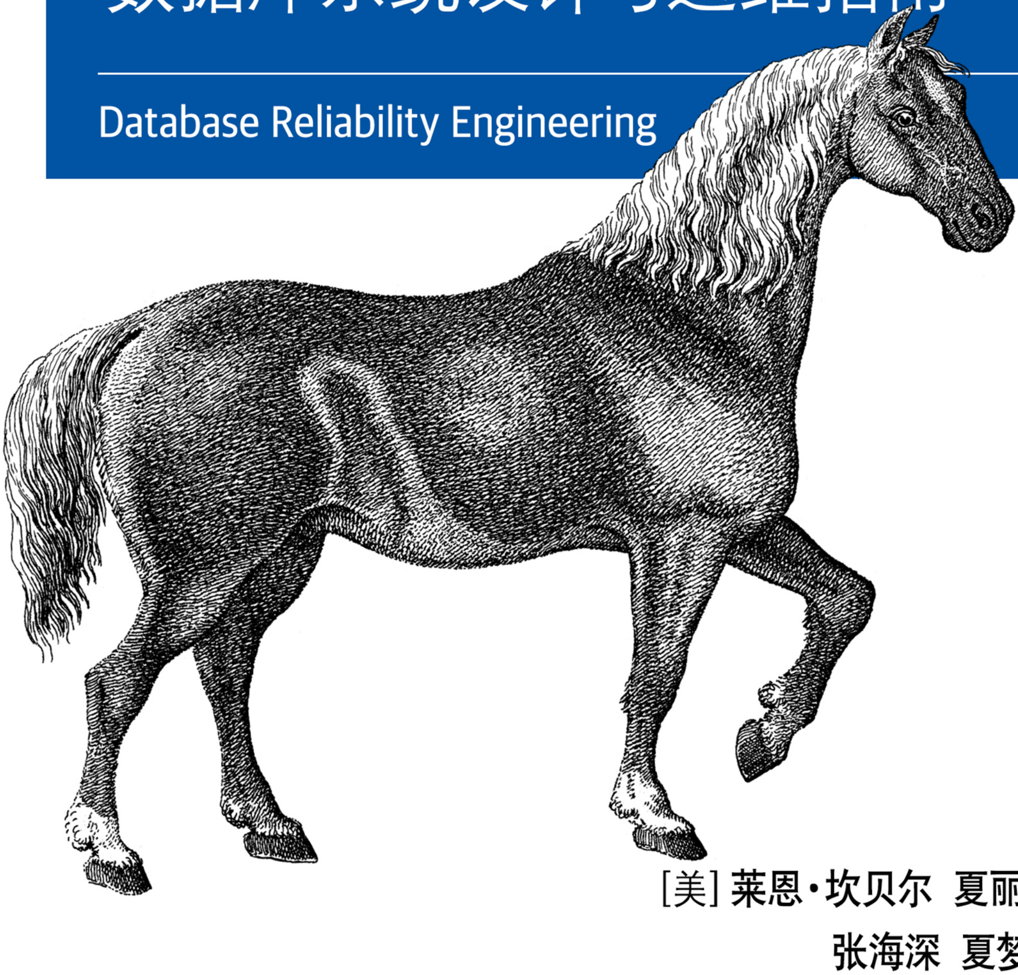
O'REILLY®

TURING 图灵程序设计丛书

数据库可靠性工程

数据库系统设计与运维指南

Database Reliability Engineering



[美] 莱恩·坎贝尔 夏丽蒂·梅杰斯 著

张海深 夏梦禹 林建桂 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

张海深

曾就职于京东、EMC、亚马逊等一线互联网公司，高级软件架构师，拥有十余年开发和管理经验，熟悉分布式存储、中间件、区块链等领域。

夏梦禹

北航硕士，在京东云担任分布式存储工程师，此前主要从事后端架构、微服务框架相关工作。涉及领域：高性能、高可用Web系统架构，性能优化，分布式KV系统。

林建桂

毕业于北京邮电大学，先后就职于京东、EMC、绿盟科技等一线科技公司，担任测试开发工程师。目前致力于提高分布式存储系统的质量和降低发布风险。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵程序设计丛书

数据库可靠性工程

数据库系统设计与运维指南

Database Reliability Engineering

[美] 莱恩·坎贝尔　夏丽蒂·梅杰斯 著
张海深　夏梦禹　林建桂 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

数据库可靠性工程 : 数据库系统设计与运维指南 /
(美) 莱恩·坎贝尔 (Laine Campbell) , (美) 夏丽蒂·
梅杰斯 (Charity Majors) 著 ; 张海深, 夏梦禹, 林建
桂译. -- 北京 : 人民邮电出版社, 2020.11
（图灵程序设计丛书）
ISBN 978-7-115-54886-3

Ⅰ. ①数… Ⅱ. ①莱… ②夏… ③张… ④夏… ⑤林…
Ⅲ. ①数据库系统—研究 Ⅳ. ①TP311.13

中国版本图书馆CIP数据核字(2020)第177990号

内 容 提 要

数据是当今企业的宝贵资产，设计、构建并维护数据存储的重要性不言而喻，而技术的快速迭代给数据库可靠性带来了更多挑战，也向数据库管理员提出了更高要求。本书从可靠性这一视角就数据库的构建和运维展开细致讨论。全书内容分为两大部分：前一部分介绍运维基础知识，包括行为准则、服务等级管理、风险管理和运维可见性；后一部分深入研究数据本身，包括基础设施、备份和恢复、安全、存储和复制、架构等。

本书适合所有对设计、构建和运维高可靠数据存储系统感兴趣的人士阅读。

◆ 著　　　　[美] 莱恩·坎贝尔　夏丽蒂·梅杰斯
　译　　　　张海深　夏梦禹　林建桂
　责任编辑　岳新欣
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　https://www.ptpress.com.cn
　北京　　　　　印刷

◆ 开本：800×1000　1/16
　印张：13.25
　字数：289千字　　　　　　2020年 11 月第 1 版
　印数：1 – 2 500册　　　　2020年 11 月北京第 1 次印刷

著作权合同登记号　图字：01-2020-3746号

定价：89.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东市监广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

> “O’Reilly Radar 博客有口皆碑。”
>
> ——*Wired*

> “O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”
>
> ——*Business 2.0*

> “O’Reilly Conference 是聚集关键思想领袖的绝对典范。”
>
> ——*CRN*

> “一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”
>
> ——*Irish Times*

> “Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”
>
> ——*Linux Journal*

目录

序

总的来说，数据库行业正经历前所未有的变革和颠覆。技术的快速迭代迫使我们不断思考，这既是挑战，也是机遇。

数据库架构飞速发展，以至于日常执行的任务变得没那么重要了，且我们投入大量精力学习的相关技能很快就落伍了。在安全、基础设施即代码以及云技术（基础设施即服务和数据库即服务等）方面涌现的创新和压力，让我们（也要求我们）重新思考如何构建数据库系统。

相应地，我们已经从传统的管理工作转向了强调架构、自动化、软件工程、持续集成、持续交付以及系统测试的工作。与此同时，我们一直保护和关心的数据，其价值和重要性已经上升了一个数量级（甚至更多），且未来会继续提升。身处这个时代，我们有幸能为世界做出有意义的重要改变。

毫无疑问，许多人曾自视优秀的数据库管理员，现在却面临不堪重负或落伍的风险。与此同时，进入该领域的新人渴望了解组织范式。这两个问题的答案是相同的：享受学习的乐趣，提升自我，保持乐观、热情和自信；尽管存在不可避免的痛苦和陷阱，但这些都是完成任务所必须经历的。这本书是一项非凡的成就，它介绍了一种思考数据库架构工程和操作的新方式；这是一本指南，涵盖了我们过去所做的一切，并把它重塑为一种新思维：数据库可靠性工程。

——Paul Vallée，Pythian 公司总裁兼首席执行官

前言

本书将介绍数据库专业人员未来的发展方向：**数据库可靠性工程师**，并讨论人们对数据库管理员工作内容的先入之见。任何与数据库管理员交互过的软件工程师或系统工程师都可能有很多这样的先入之见。

传统上，数据库管理员充分理解数据库的内部结构。他们每日与程序优化和查询引擎打交道，并以设计和调整高性能、专业化的系统为业。当需要运用其他技能来让数据库更好地运行时，他们便去学习，例如学习如何在 CPU 或磁盘之间分配负载，如何配置数据库以提高 CPU 性能，以及如何评估存储子系统。

当遇到可见性问题时，数据库管理员会学习如何为关键指标构建图表；当遇到架构限制时，他们会去了解缓存层；当遇到单点瓶颈时，他们会学习（并帮助推动开发）新的设计模式，比如分片。其间他们掌握了新的操作技术，例如缓存失效、数据重新分布和滚动升级数据库变更等。

长期以来，数据库管理员总是忙于自己的事。各岗位所用的工具不同，硬件不同，语言也不同：数据库管理员用 SQL，系统工程师用 Perl，软件工程师用 C++，Web 开发人员用 PHP，网络工程师则在打造自己的工具。只有一半的团队以某种方式使用过版本控制系统，而且他们不会谈论或干涉彼此的领域。这是为何？因为那样就会踏入陌生领域。

这种模式有效和可持续的好日子快到头了。本书通过数据库工程师的视角来展现**可靠性工程**。本书无意涵盖所有内容，而是描述对你的职业生涯重要的事情。而且，该框架适用于各种数据存储、架构和组织。

写作初衷

本书是我们近 5 年工作的经验总结。莱恩没有接受过任何正式的技术培训就担当了数据库管理员。她既不是软件工程师，也不是系统管理员，而是从音乐和戏剧行业转入技术领域

的。数据库的结构、和谐、对位和编排的思想深深吸引了她。

自那时起，她便开始学习数据库知识，并与近百名数据库管理员一起工作过。数据库人员的背景五花八门，有些有软件背景，有些有系统背景，有些甚至来自数据分析领域和商业领域。然而，优秀的数据库管理员对公司数据的安全性和可用性始终怀抱热情，以为己任，以无上的激情和超高的工作强度履行职责。此外，数据库管理员也在软件工程师和系统工程师之间扮演关键角色。由于涉及各个领域，数据库管理员也被视为最初的 DevOps 工程师。

夏丽蒂一直在创业公司从事运维工作。她有丰富的创业公司工作经历，擅于快速启动基础设施、做出关键决策、承担风险以及基于有限资源做出艰难选择，并屡获成功。一个偶然的机会，喜欢数据的她成为了数据库管理员。因为之前她所在的运维团队没有专门的数据库管理员，所以开发团队和运维团队最终承担了数据库管理员的工作。

我们基于长期的工作经历以及不同的职业背景，重新审视并拥抱近十年的变化。数据库管理员的工作通常艰辛而默默无闻。现在，有了方法和公众支持来把这个角色“点亮”，让数据库管理员可以专注于创造更多价值。

本书力图延续前几代人的影响，帮助下一代工程师享受职业生涯，取得更多成就。

读者对象

本书适合所有对设计、构建和运维可靠性数据存储系统感兴趣的人士。无论是希望拓展数据库相关知识的软件工程师或系统工程师，还是希望进一步提升技能的数据库专业人员，都能从本书获益。行业新人也能通过阅读本书加深理解，毕竟本书讲的是框架。

本书假设你基本掌握了 Linux/UNIX 系统管理以及 Web 或云架构的基础技术，对其他专业（系统管理或软件工程）有一定了解，想拓展技能（数据库软件技术）；或者你正处于职业生涯的初期或中期，希望朝着数据库专家的方向深化技术。

如果你是管理人员，甚至是项目管理人员，阅读本书可以了解服务对数据存储的需求。我们坚信，管理人员需要理解数据库的原理和运维，以助力团队和项目取得成功。

你可能没有传统的技术背景。也许你是“半路出家”的数据库管理员，曾经是业务分析师，进入了数据库的知识海洋来学习如何使用数据库。很多数据库专业人员是通过 Excel 而不是开发或系统方面的工作踏入数据库领域的。

内容安排

本书分为两大部分：前一部分是运维核心课程，讲解数据库工程师、软件工程师，甚至产

品负责人都应该掌握的运维基础知识；后一部分将深入研究数据本身，包括建模、存储、复制、访问等，还会讨论架构选型和数据流水线技术。

有这样一种说法：如果你不是优秀的工程师，就不是优秀的可靠性工程师，更不可能成为优秀的数据库可靠性工程师。现代数据库可靠性工程师专门研究系统工程基础之上的数据特定领域问题。

重点是，**任何工程师都可以运行数据服务**。我们有共同的语言，使用相同的代码库和相同的代码审查流程。运维数据库是运维工程的延伸（像把特定知识和认知的奶油糖霜按比例覆在运行一定规模系统的蛋糕上），正如杰出的网络工程师不仅要知道如何成为工程师，还要知道如何处理流量、应该担心出现什么状况、当前的最佳实践、如何评估网络拓扑结构等。

每章的内容简介如下。

第 1 章介绍数据库可靠性工程的概念。从指导原则开始，然后过渡到运维核心，最后基于马斯洛的需求层次理论介绍建立数据库可靠性工程师愿景的框架。

第 2 章介绍服务等级的要求，这和产品的特性要求同样重要。这一章将讨论什么是服务等级的要求及其定义方式，这一内容并不像听起来那么简单。然后将讨论随着时间的推移如何衡量和处理这些要求。

第 3 章讨论风险评估和管理。简单讨论风险后，介绍将风险评估纳入系统和数据库工程的实际流程，并且讲解其中的陷阱和复杂性。

第 4 章讨论运维可见性。这一章会讨论度量值和事件、如何设置度量指标，以及如何随着时间迭代。我们将深入讲解监控系统的组件及其客户端。

第 5 章和第 6 章深入讨论基础设施工程和管理，包括为数据存储服务构建主机的原则。我们将深入研究虚拟化和容器化、配置管理、自动化、编排，帮助你理解构建存储系统所需的全部动态组件。

第 7 章介绍备份和恢复。这也许是数据库工程师需要掌握的最关键的技术。数据丢失意味着一切都完了。从服务等级的要求开始，我们评估恰当的备份和恢复方法，以及如何扩展和测试这一在运维中非常重要却容易被忽略的操作。

第 8 章讨论发布管理。这一章将介绍如何对数据存储服务进行测试、构建和部署变更，以及如何更改数据访问代码和 SQL。部署、集成和交付是这一章的核心内容。

第 9 章讨论数据安全。数据安全关乎公司存亡。这一章将讲解在不断演进的数据库基础设施中规划和管理安全性的策略。

第 10 章讨论数据存储、索引和复制。这一章将介绍关系数据是如何存储的，然后将其与有序字符串和 LSM 树进行比较。在回顾各种索引之后，将探索数据复制拓扑。

第 11 章是数据存储领域指南，将讨论评估或操作的数据存储的各种属性，其中包括对应用程序开发人员和架构师而言都非常重要的概念属性，以及侧重于数据存储物理实现的内部属性。

第 12 章介绍分布式数据库的常用架构模式，以及其中用到的数据流水线技术。首先会介绍数据库生态系统中典型的架构组件及其优点、复杂性和一般用法，然后介绍架构和数据流水线技术，并给出少量示例。

第 13 章介绍如何在组织中建设数据库可靠性工程文化，探讨在当今情形下，从传统管理员转变为数据库可靠性工程师的多种方法。

排版约定

本书使用以下排版约定。

- **黑体**

  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 等宽粗体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示普通的注记。

该图标表示警告或警示。

O’Reilly在线学习平台（O’Reilly Online Learning）

近 40 年来，O’Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O’Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O’Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 https://oreilly.com。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表[1]、示例代码以及其他信息。本书的网站地址是：http://bit.ly/database-reliability-engineering。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：https://facebook.com/oreilly。

请关注我们的 Twitter 动态：https://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：https://www.youtube.com/oreillymedia。

注 1：可以访问本书图灵社区页面（https://www.ituring.com.cn/book/2103）查看或提交中文版勘误。——编者注

电子书

扫描如下二维码，即可购买本书中文版电子版。

第1章

数据库可靠性工程介绍

本书旨在为读者提供指导与框架，助你成为优秀的数据库可靠性工程师。在定书名时，我们选择了面向**可靠性工程师**，而非管理员。

谷歌工程副总裁本·特雷诺就可靠性工程发表了如下看法：

> 基本上是以前由运维团队完成的工作，换成拥有软件专业知识的工程师来做，并且指望其有意愿和能力用自动化代替人工劳动。

如今的数据库专业人员必须是工程师，而不是管理员。我们构建、创造新事物。作为实践 DevOps 的工程师，我们一起工作，共担责任。作为工程师，我们利用可复用的流程、掌握的知识和积累的经验，来设计、构建和操作生产级数据存储及其数据结构。而作为数据库可靠性工程师，我们必须进一步深入掌握操作原则和数据库专业知识。

如果了解当今基础设施的非存储组件，就会发现通过编程（通常是自动的）方式，易于构建、运行以及销毁系统。这些组件的生命周期较短，以天计，有时甚至以小时或分钟计。当一个组件下线之后，就会启动其他许多组件，以保证服务质量。

我们的下一个目标是，在可靠性工程和 DevOps 文化的范式下，为设计、构建和维护数据存储确立指导原则和实践框架。无论你所在的企业处于哪个发展阶段，都可以把这些知识应用于任何数据库技术或环境中。

1.1 数据库可靠性工程师的指导原则

我们当初动笔的时候，问自己的第一个问题是：数据库行业这种新的迭代逻辑背后的原则是什么？如果重新定义设计、管理数据存储的方式，就需要定义受推崇的行为准则。

1.1.1 保护数据

传统上，保护数据一直是数据库专业人员的基本原则，现在仍是如此。以下是普遍接受的做法。

- 软件工程师和数据库工程师的职责严格分离。
- 有严格的备份和恢复流程，并定期测试。
- 有规范的安全流程，并定期审计。
- 有昂贵的数据库软件，并且持久、可靠。
- 有昂贵的底层存储系统，并且所有组件都有冗余。
- 大范围的变更管控和操作管理。

在具有协作文化氛围的团队中，职责严格分离不仅是一种负担，还会限制创新和速度。第8章将讨论创建安全网的方式，以此应对职责分离的问题。此外，这些环境更关注测试、自动化和减轻影响，而不是大范围的变更管控。

架构师和工程师比以往更倾向于选择开源数据存储，而这些数据存储不像 Oracle 那样能保证持久性。这种非严格的持久性，有时会给业务快速发展的团队带来所需的性能优势。第11章将讨论如何选择正确的数据存储方式，以及这些选择的影响。了解数据处理工具，并能有效地选择合适的工具，属于必备技能。

底层存储也发生了重大变化。在虚拟化时代，网络和临时存储在数据库设计中占有一席之地，第5章将进一步讨论这个问题。

构建在临时存储之上的生产数据库

2013 年，Pinterest 将 MySQL 数据库实例迁移到 AWS（亚马逊云服务）的临时存储上。临时存储实际上意味着，如果计算实例发生故障或关闭，磁盘上存储的所有东西都将丢失。Pinterest 之所以选择临时存储，是因为它吞吐量可观且延迟较低。

要做到这一点，需要在自动化、完善的数据备份和恢复，以及应用程序方面大量投资，以容忍在重建节点时导致集群失效。临时存储不允许快照，这意味着不能通过快照前滚事务日志的方式进行恢复，而要通过网络复制完整的数据库。

这表明，运用正确的流程和工具，可以在临时存储的环境中保证数据的安全性。

数据保护的新方法大致如下。

- 跨职能团队共担责任。
- 由数据库可靠性工程师提供支持的标准化、自动化备份和恢复过程。
- 由数据库可靠性工程师和安全团队提供支持的标准化安全策略和流程。
- 自动化配置和部署应用的全部策略。
- 数据需求决定数据存储，对持久性需求的评估将成为决策过程的一部分。
- 依赖自动化过程、冗余和经过反复检验的良好实践，而不是昂贵、复杂的硬件。
- 在部署和基础设施自动化过程中实现变更，重点关注测试、回退和减轻影响。

1.1.2 大量自助服务

到目前为止，有才干的数据库可靠性工程师比 SRE（site reliability engineer，网站可靠性工程师）更稀有，大多数公司雇不起两位以上这样的员工。因此，我们必须通过为团队创建自助服务平台，创造尽可能多的价值。制定标准并提供工具，以便团队能够顺利部署新服务，并按照所需的速度适当地更改，而无须依赖数据库工程师超负荷工作。自助服务的例子包括：

- 提供合适的插件，确保从数据存储中收集适当的度量值；
- 构建备份和恢复工具，使其可以部署到新的数据存储中；
- 为数据存储定义参考架构和配置，这些数据存储被批准用于操作，并且可以由团队部署；
- 和安全团队一起定义数据存储部署的标准；
- 为数据库变更集构建安全的部署方式和测试脚本。

换言之，高效能数据库可靠性工程师的职责是授权和指导他人，而不是充当“守卫”。

1.1.3 消除琐事

谷歌 SRE 团队经常使用短语“消除琐事”（elimination of toil），《SRE：Google 运维解密》一书的第 5 章对此做了讨论。该书将“琐事”定义如下：

> 与运行生产服务相关的工作，往往是手动的、重复性的、自动化的、策略性的、缺乏持久价值的，并且随着服务的增长而线性扩展。

有效的自动化和标准化是必要的，可以确保数据库可靠性工程师不会困于琐事。本书将列举数据库可靠性工程师所面临的一些琐事及其缓解之法。尽管如此，“琐事”这个词的含义仍然是模糊的，有很多先入之见，且因人而异。本书讨论“琐事”时，着重谈论重复性、非创造性和非挑战性的人工工作。

手动变更数据库

在许多客户环境中，数据库工程师被要求审查和实施数据库变更，例如修改表或索引，添加、修改或删除数据，等等。所有人都安心于数据库管理员正在实施这些变更，并实时监控变更的影响。

在客户现场，变更的频率非常高，并且这些变更常常是有影响的。最终数据库管理员每周花 20 个小时对整个环境实施滚动变更。毫无疑问，花半周的时间来执行这些重复性的任务，难免让人心生不满，意欲放弃。

面对资源短缺，管理层最终允许数据库团队构建一个滚动变更 schema 的自动化工具。一旦数据库工程师审查并批准了变更集，软件工程师就可以运行该程序实施数据库变更了。之后所有人都能放心地使用工具引入变更并监控，从而让数据库可靠性工程师团队有更多时间来把这些流程集成到部署平台。

1.1.4 数据库并不特殊

我们的系统与满足业务需求的其他组件一样重要。我们必须努力实现标准化、自动化和弹性。对此至关重要的是，数据库集群的组件并非神圣的。有时我们会移除某些组件，毫无顾虑地进行替换。玻璃房里脆弱的数据存储已成历史。

“宠物”和“牛”的比喻通常用于描述定制的服务组件和通用服务组件之间的区别，这个比喻来自微软杰出的工程师比尔·贝克。“宠物”服务器是指当它们出现异常时，你需要精心照顾以使其恢复健康。它们也有名字。2000 年在 Travelocity 时，我们的服务器以动画片《辛普森一家》中的角色命名，运行 Oracle 的两台 SGI 服务器分别称作 Patty 和 Selma。我在深夜花了很多时间处理这两台服务器的问题，它们的维护成本很高。

“牛”服务器上有数字，但没有名字。不需要花时间定制服务器，更不用说登录各个主机了。当它们显现“生病”（异常）的迹象时，就把它们从“牛群”（集群）中剔除。当然，如果你发现患病的“牛”达到一定的数量时，就应该把那些待宰的“牛”留给“兽医”检查。但是，不必把这个比喻搞得更复杂。

数据存储是拥有“宠物特质”的最后一部分系统。毕竟它们拥有“数据”，所以不能简单地将它们视为寿命短、标准化程度高、可替代的“牛”。那么副本的特殊规则呢？从服务器的不同配置呢？

1.1.5 消除软件和运维之间的障碍

基础设施、配置、数据模型和脚本都是软件的组成部分。我们要像其他工程师那样，学习并采用软件生命周期的方式进行管理：编码、测试、集成、构建、测试和部署。我们提到测试了吗？

对于熟悉运维和脚本的人而言，这可能是最艰难的范例转换。在软件工程师引导组织、系统和服务以满足需求时，组织内可能存在**阻抗不匹配**（impedance mismatch）。软件工程组织有非常明确的方法来开发、测试、部署功能和应用程序。

在传统环境中，设计、构建、测试以及部署基础设施和相关服务的基本流程，在软件工程、系统工程和数据库管理员之间是分裂的。之前讨论的范式转换正在消除这种不匹配，这意味着数据库可靠性工程师和系统工程师需要使用相似的方法来完成工作。

> **软件工程师必须学习运维**
>
> 运维人员经常被告知“学习编程，否则辞退”。我认同这一点，但软件工程师也必须学习运维。那些没有学习运维和基础设施原理的软件工程师，编写的代码往往脆弱、低效，甚至不安全。只有所有团队水平相当，阻抗不匹配才会消失。

数据库可靠性工程师也可能发现自己直接融入了软件工程团队，在相同的代码库中工作，检查这些代码如何与数据存储交互，修改代码以优化性能、完善功能、增强可靠性。消除这种团队阻抗之后，相较于传统模型，可靠性、性能和速度会提升一个数量级，并且数据库可靠性工程师必须适应这些新流程、文化和工具。

1.2 运维核心概述

运维是数据库可靠性工程师的核心技能之一，是设计、构建、测试和维护具有一定规模且要求高可靠性系统的基石。这意味着如果你想成为数据库工程师，就需要知道这些。

宏观层面的运维不是一个角色。运维是企业围绕发布与维护高质量系统和软件，所积累的技能、知识和价值观的总和。它既是隐性价值观，也是显性价值观、习惯、团队知识和奖励体系。从技术支持人员到产品人员，再到首席执行官，共同参与运维。

这一点通常做得不好。许多公司的运维文化很糟糕，令人生厌。这可能会影响声誉，不管是在系统、数据库还是网络方面，许多人在想到运维工作时就会想到这一点。尽管如此，运维文化仍然是组织如何执行技术任务的关键。如果说哪家公司完全没有运维，肯定不可信。

也许你是软件工程师或者是基础设施和平台即服务的支持者，也许你质疑数据库工程师是否需要懂运维，并认为 Serverless 可以让软件工程师无须思考或关心运维工作，这种想法是完全错误的。事实恰恰相反，这是一个没有运维团队的美丽新世界——为你做运维工作的是谷歌 SRE、AWS 系统工程师、PagerDuty 和 DataDog 等。在当今世界，应用工程师需要在运维、架构和性能方面做得更好。

1.3 需求层次

你可能在大企业或创业公司工作过。当接触并研究系统时，有必要考虑一下，如果让你承担运维数据库系统的工作，第一天你会做些什么。有备份吗？备份能正常工作吗？你确定吗？是否有可以进行故障转移的副本？你知道如何进行故障转移吗？备份的电源、路由器、硬件或可用区与主服务器相同吗？当备份工作不正常时，你能发现吗？如何发现？

换言之，我们需要讨论数据库需求的层次结构。

按照马斯洛的需求层次理论，人类的欲望像一座金字塔，由下至上分别是：生存、安全、爱和归属感、尊重以及自我实现，而只有满足需求，人类才能蓬勃发展。金字塔的底部是最基本的需求，比如生存。每一层的需求都是进入更高一层的条件——在获得安全感之前要先满足生存需求，在获得爱和归属感之前要先满足安全需求，以此类推。一旦满足了下面 4 个层次的需求，人类就达到了自我实现，可以安全地探索、游戏、创造和充分发展潜能。这就是它对于人类的意义。下面用这个比喻来说明数据库需要什么。

1.3.1 生存和安全

数据库最基本的需求是备份、复制和故障转移。你有数据库吗？数据库运作正常吗？可以 ping 通吗？应用程序有响应吗？有备份吗？恢复有效吗？你怎么知道它工作不正常？

你的数据安全吗？是否有多个可用副本？你知道怎么进行故障转移吗？你的副本是分布在多个可用区、多个电源板和机架上吗？各个备份是否一致？你能恢复到某个时间点吗？你能发现数据有损坏吗？如何发现呢？第 7 章将深入探讨这些内容。

这也是开始准备扩展的时候。过早地扩展不可取，但在确定关键数据对象的 ID、存储系统和架构时，应该考虑**分片**、增长和扩展。

> **扩展模式**
>
> 本书会经常提到扩展。扩展性是系统或服务应对负载不断增加的能力。这可能是**真实的**能力，因为已经部署了所有支持数据增长的部件；也可能是**潜在的**能力，因为处理组件和资源增加的基石已经铺好。一般而言，可以通过以下 4 种途径实现扩展。
>
> - 通过分配资源实现垂直扩展，也称**垂直扩展**。
> - 通过复制系统或服务实现水平扩展，也称**水平扩展**。
> - 将工作负载分成较小的功能集，让每个工作单元能够独立扩展，也称**功能分区**。
> - 将特定工作负载分成相同的分区，而不是分成正在处理的特定数据集，也称**分片**。
>
> 第 5 章将介绍上述模式的具体细节。

1.3.2 爱和归属感

爱和归属感意味着，在软件工程过程中，把数据当作一等公民，打破数据库和其他系统之间的竖井（silo）。这既是技术上的，也是文化上的，而这就是可以称其为“DevOps 需求”的原因。从高层次上讲，这意味着要像管理其他系统一样管理数据库，也意味着在文化上鼓励流动性和跨职能。在爱和归属感阶段，你会逐渐停止登录系统和以 root 身份执行命令。

在该阶段，你们会采用相同的代码审查流程和部署实践。数据库基础设施和配置应该采用与其他所有架构组件相同的流程。与数据打交道，应该和跟应用的其他部分打交道一样，这会鼓励所有人参与其中并支持数据库环境。

克制向开发人员灌输恐惧的冲动。这很容易做到，也很有诱惑力，因为一切尽在掌控的感觉非常好。但事实并非如此，而且你也无法掌控一切。如果你把精力放到构建“护栏”上，以防止任何人意外破坏，对所有人来说会更好。培养和授权所有人对自己的变更负责。不要再提杜绝故障，因为这是不可能的。换言之，创建有弹性的系统，并且鼓励所有人使用数据存储。

Etsy 的“护栏”

Etsy 引入了一个名为 Schemanator 的工具来实施数据库变更（或者说变更集），这对生产环境来说是非常安全的。多个“护栏”的存在能让软件工程师直接实施数据库变更。这些“护栏”如下所示。

- 变更集的探索性审查，以验证 schema 设计是否遵循了相应规范。
- 变更集的测试，以验证脚本能否执行成功。
- 预先检查，让工程师知晓集群的当前状态。
- 滚动升级，对“离线”的数据库执行有影响的变更。
- 把工作流分解为子任务，以便在发生意外时可以取消。

1.3.3 尊重

尊重处于需求金字塔的次顶端。对人类来说，这意味着尊重和掌控；对数据库来说，这意味着可观测性、可调试性、自我检查和可探测性。关键是，不仅要了解存储系统本身，还要能关联相关事件。同样，该阶段包含两个方面：一是当前阶段生产服务的演进方式，二是人员。

服务本身应该能表明它是运行正常还是宕机或者出错了，而无须你查看监控图。随着服务的成熟，由于系统演变轨迹变得更加可预测，因此变更速度会变慢。由于存储系统在生产环境中运行，所以你对其弱点、行为以及故障状况的了解会日益加深，这类似于数据基础

设施的青年时期。其间你最需要的是能够了解正在发生什么。产品越复杂，其动态部件就越多，也就需要投入更多精力开发工具来搞清楚状况。

还需要有“旋钮”来降级服务质量，避免服务彻底宕机，比如：

- 将站点标记为只读；
- 禁用某个特性；
- 将写请求排队，延后处理；
- 将恶意者或者特定端点拉进黑名单。

人员的需求相似，但并不完全相同。一种常见的情况是，他们对生产环境反应过度。他们对可能出现的问题没有清晰的认识，所以试图监控一切指标，以致草木皆兵。从没有监控图发展到成百上千的监控图（99% 是完全没有意义的）很简单，但这并非好事，实际上可能适得其反。如果这导致了更多的噪声，你的人员将无法找到问题的根源，而只能跟踪日志文件并猜测，这与没有监控图一样糟糕，甚至更糟糕。

这时，你会打断他们，唤醒他们，培训他们不要在意收到的警报或对其采取行动，这会耗尽他们的精力。在早期阶段，如果你希望所有人都随叫随到，就需要把事情文档化。当你处于起步阶段、随叫随到、迫使人们走出舒适区时，给他们一点帮助。编写简洁有效的文档和流程。

1.3.4 自我实现

就如每个人最好的自己是独一无二的，每个组织实现的存储层也是独一无二的。 适合 Facebook 的存储系统，也许并不适合 Pinterest 或 GitHub，更不用说小型创业公司了。但是，就像健康、自律的人有自己的行为模式（他们不会在杂货店乱发脾气，保持健康饮食并且锻炼身体），健康、自我实现的存储系统也有类似的范式。

在这种情况下，自我实现意味着数据基础设施能帮助你实现目标，而且数据库相关工作流程不会妨碍进度。相反，它们能帮助开发人员完成工作，并帮助他们避免一些错误。常见的运维痛点和令人厌烦的故障应该由系统自我修复，系统能保持健康状态，而无须人工介入。这意味着有满足系统需求的扩容方案，无论是每过几个月就需要扩容 10 倍，还是系统要稳定运行 3 年之后才扩容。显然，如果数据库基础设施成熟、可靠，你就可以把更多时间花在思考其他事情上，比如创造新产品或者预料未来的问题，而不是处理当前的问题。

所处的需求层级随着时间的推移上下变动是正常的。这些层级作为框架，主要用于帮助思考问题的优先级，比如确保有可用的备份要比写脚本来动态重新分片和扩容重要得多。如果线上数据只有一个副本，又或者你不知道当主服务器宕机时如何进行故障转移，那么你应该停止手头工作，优先解决这个问题。

1.4 小结

数据库可靠性工程师是从现有的、众所周知的角色演变而来的。重要的是，本书给出的框架能让我们在纷繁多变的当今世界中，重新思考管理数据存储的职责。后文将详细探讨这些职责，优先讨论运维职责，因为这在数据库工程的日常工作中十分重要。勇敢前行吧，无畏的工程师们！

第 2 章

服务等级管理

要成功地设计、构建和部署服务，首先需要了解服务的预期目标。本章介绍服务等级管理的定义及其内容，然后讨论如何定义服务的预期目标，以及如何监控和报告以确保服务符合预期。本章将通过构建一系列健康的服务等级需求来解释该过程。

2.1 为何需要SLO

设计和构建的服务对其运行时特征有一系列要求，这通常称为 SLA（service-level agreement，服务等级协议）。SLA 不仅是需求列表，还包含补救措施、影响等内容（这些内容超出了本书的范畴）。本书将着重探讨 SLO（service-level objective，服务等级目标），SLO 是架构师和运维人员在指导系统的设计和运维时所做的承诺。

服务等级管理颇具难度，一章恐难以尽述，重点是理解其中的微妙之处。下面通过几个例子说明为何这个问题很难。

- 你可能会说："我只报告 API 成功处理的请求的百分比。"好吧，报告谁的？ API 的？显然有问题，如果负载均衡服务宕机了呢？或者服务发现系统探明某个数据库服务不可用，而返回 200 错误呢？
- 也许你会说："好的，我们将采用第三方端到端的检查，并计算正确读写数据的请求数量？"这很不错，端到端检查是更可靠的报警方式，但是，需要对所有后端服务都这样做吗？
- 你会在 SLO 中考虑不那么重要的服务吗？用户可能希望 API 服务的可用性为 99.95%，批处理产品的可用性为 97%，而不是 API 和批处理产品的可用性均为 99.8%。

- 你对客户端有多大的控制权？如果你的 API 服务的可用性为 98%，但客户端会自动重试，并且在 3 次重试中确保响应率达 99.99%，那么用户可能就不会发现问题。哪个数才能精确描述服务可用性呢？
- 可能你会说“我只统计错误率”，但是，如果错误是由用户发送无效或格式错误的请求而导致的呢？实际上，你对此束手无策。
- 也许你的服务可用性达到了 99.999%，但是在其中 15% 的时间，响应延时超过 5 s。这可接受吗？根据用户的行为，这实际上可能意味着网站对某些用户请求没有响应。从技术上讲，你自己统计的服务可用性为 99.999%，但是用户会非常不满，而且这也无可非议。
- 如果对于 98% 的用户而言，网站的可用性为 99.999%，而对于另外 2% 的用户仅能提供 30%~70% 的可用性呢？这种情况该如何精确计算？
- 如果一个数据分片或一个后端服务宕机或者变慢呢？如果升级过程中出现 bug，导致 2% 的数据丢失怎么办？如果经历了一整天的数据丢失，但仅针对某些表，怎么办？如果用户从来没有发现数据丢失（因为数据的自然属性），但是你报告说有 2% 的数据丢失，导致所有人把数据迁移至其他平台怎么办？如果这 2%“丢失的数据”中实际上包含因地址重写而找不到的数据，但数据并未真的丢失呢？
- 如果有些用户因为 Wi-Fi 信号很差、线缆老化，而只能体验到 95% 的可用性呢？或者客户端到服务端的路由配置很糟糕呢？你要为此负责吗？
- 如果用户来自全国不同地区呢？那他们很可能会责怪你（比如某些网络提供商的 DNS 服务的 UDP 包超过阈值——你可以解决这个问题）。
- 如果你得出的可用性是 99.97%，但是每个错误都会导致整个网站无法加载呢？如果你得出的可用性是 99.92%，但是由于一个页面有 1500 个组件，当某个小组件加载失败时，用户几乎不会发现呢？哪种体验更好？
- 统计实际的错误率和按照时间分片来统计，哪种方式更好？当错误或超时请求数超过某个阈值就按照分钟（或秒）来统计？

> **5 个 9**
>
> 很多人使用“几个 9”这样的简写方式来描述可用性，例如系统可用性为“5 个 9”意味着构建的服务在 99.999% 的时间是可用的。同理，“3 个 9”表示可用性为 99.9%。

这就是为什么随着时间的推移，设计、管理和调整 SLO 以及可用性度量的实践与其说是计算问题，不如说是社会科学问题。如何计算一个可用性，以准确反映用户真实体验、建立信任并推动朝正确的方向前进？

从团队的角度看，无论你们一致认为重要的可用性度量值是什么，在某种程度上都会成为不真实的数字，即使只是下意识的。在判断服务的可靠性变高或变低时，或者在判断是否需要把资源从功能开发转移到可用性（反之亦然）时，需要关注这些数字。

从用户的角度来说，最重要的是度量值尽可能地**反映他们的真实体验**。如果能计算每个用户或者分片的度量值，并按照任意维度划分数据，甚至是像 UUID 这样高基数的指标，将是非常有用的。Facebook 的 Scuba 和 Honeycomb 就是这么做的。

2.2 服务等级指标

在评估 SLO 的需求时，通常会考虑一组有限的**指标或度量值**，我们称其为服务等级指标（service-level indicator，SLI），并基于这些指标设定需求。其中，我们既会考虑理想参数，也会考虑实际的工作参数。可以将 SLO 视为一个或一组定义服务预期目标的指标，这通常是因为这些指标之间存在内在联系。

例如，延时在超过特定值后会变成可用性问题，因为系统实际上不可用。不考虑吞吐量的延时很可能不准确，不一定能准确反映负载下系统的真实状况。下面列举并解释典型指标。

2.2.1 延时

延时，也称“响应时间”，是一个基于时间的度量，表明收到一个请求后需要多长时间响应。最好是测量端到端的响应延时，而不是度量组件之间的延时。这是以用户为中心的设计，对于任何拥有用户的系统（任何系统都有用户）都至关重要。

> **延时和响应时间的对比**
>
> 关于延时和响应时间的区别有很多争论。一些人认为延时是指请求到达服务所花的时间，而响应时间是指到服务处理请求所花的时间。在本书中，延时指从请求发起到响应数据总的往返时间。

2.2.2 可用性

通常用服务预期可用的时间比例来表示可用性。可用性定义为客户端的请求返回预期响应的能力。注意，这里没有提到时间，这就是为什么大多数 SLO 中既包含响应时间，也包含可用性。延时超过特定值后，尽管请求被完全处理了，但此时的服务也被视作不可用。可用性通常用百分比表示，比如某个时间窗口下可用性是 99%。该时间窗口的所有采样都会被聚合。

2.2.3 吞吐量

另一个常用的 SLI 是吞吐量，即一段时间内被成功处理的请求量，通常以秒为单位进行测量。吞吐量和延时联用会非常有用。团队必须在最大的吞吐量目标下测量延时，否则测量没有意义。越过临界点之前，延时一直很稳定。我们必须找到吞吐量目标下的临界点。

2.2.4 持久性

持久性特定于存储系统和数据存储服务。它表示写操作被成功持久化到存储系统中，以便在随后的操作中获取。这也可以表示为一个时间窗口，例如系统发生故障时，不能丢失超过两秒的数据。

2.2.5 成本或效率

成本或效率通常被忽略，或者在服务等级的讨论中不被提及，而是被归入预算，并且常常没有得到有效的追踪。尽管如此，服务的总体成本仍是大多数业务的关键指标。理想情况下，应该以每次操作的成本来表示，例如一次网页浏览、一次订阅或者购买。

组织应该把下列行为作为服务运维的一部分。

- **新服务**

  制定 SLO，这在传统模型中可能称为运维等级协议。

- **新 SLO**

  设置适当的监控，以评估实际度量值与目标度量值的差别。

- **现有服务**

  定期审查 SLO，以验证制定的 SLO 是否考虑了当前服务的重要性。

- **实现 SLO**

  定期报告历史上的和现在的测量指标，以审视符合或违背 SLO 的状况。

- **服务问题**

  对于影响服务等级和当前状态的问题，寻找解决和修复方法。

2.3 定义服务目标

SLO 应该按照产品构建时的需求来建立。我们称之为**以用户为中心**的设计，因为我们应该根据用户需求来定义需求。通常最多只需要 3 个指标，更多指标并不能带来更多价值。指标过多通常意味着已经包含主要指标的特征了。

2.3.1 延时指标

延时 SLO 可以表示为基于某个指标的范围，例如延时必须小于 100 ms（当明确做出假设时，这实际上是 0~100 ms 的范围）。延时对于用户体验来说至关重要。

为何延时至关重要

缓慢的或者间歇性缓慢的服务会比系统宕机流失更多用户。事实上，速度相当重要。谷歌研究院发现，引入 100~400 ms 的延时，在 4 周和 6 周内的搜索率分别下降 0.2% 和 0.6%。Jake Brutlag 的“Speed Matters”一文谈到了更多细节。下面列举几个令人吃惊的数据。

- 亚马逊：延时每增加 100 ms，销售额会损失 1%。
- 谷歌：如果页面加载延迟 500 ms，会导致搜索量减少 25%。
- Facebook：加载页面时间延迟 500 ms，会导致网络流量减少 3%。
- 页面响应每慢 1 s，用户满意度就会下降 16%。

关于可用性的 SLO 可以表示为：请求延时必须少于 100 ms。

如果将下限设为 0，可能会导致某些功能异常。一位性能工程师花一周时间将响应时间缩减到 10 ms，但使用应用的移动设备很少有足够快的网络来享受优化带来的效果。换言之，这位性能工程师浪费了一周时间。可以将 SLO 迭代成：请求延时必须为 25~100 ms。

接下来思考如何收集这些数据。如果我们正在查看日志，可能会取 1 分钟的请求并计算平均数。这样做其实是有问题的，因为在大多数分布式网络系统的延时分布中，存在较小比例的异常值，但这些异常值可能相当大。这将扭曲平均值，也会对监控它的工程师隐藏完整的工作负载特征。换言之，聚合响应时间是一个**有损过程**（lossy process）。

事实上，在考虑延时的情况下，必须考虑延时的分布。延时几乎从不遵循正态分布、高斯分布或泊松分布，所以平均值、中位数和标准偏差都无意义，这还算是最好的结果，更糟糕的结果是误导他人。更多细节，可以参考 Tyler Treat 的文章“Everything You Know About Latency Is Wrong”。

为了加深理解，可以参考由 Circonus（大规模监控产品）提供的图 2-1 和图 2-2。这些图用于解释**削峰**（spike erosion），这正是我们正在讨论的现象。图 2-1 是一个在较大的时间窗口中计算平均值的图，这些平均值展示了一个月的数据。

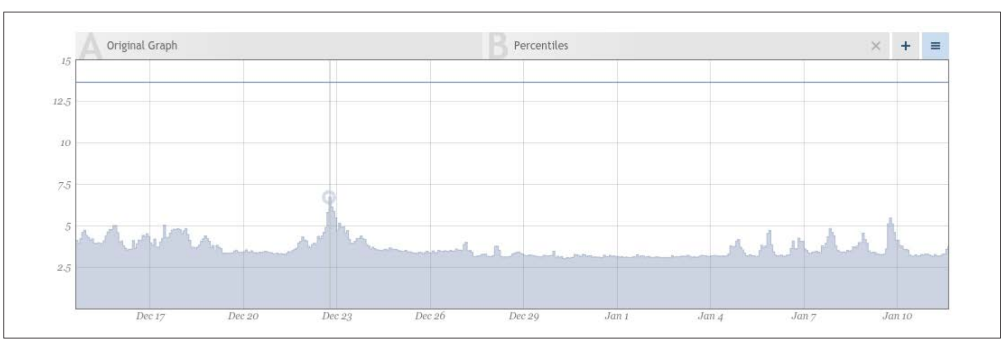


图 2-1：较大时间窗口的平均延时

图 2-2 展示了较小时间窗口（4 个小时）的平均延时。

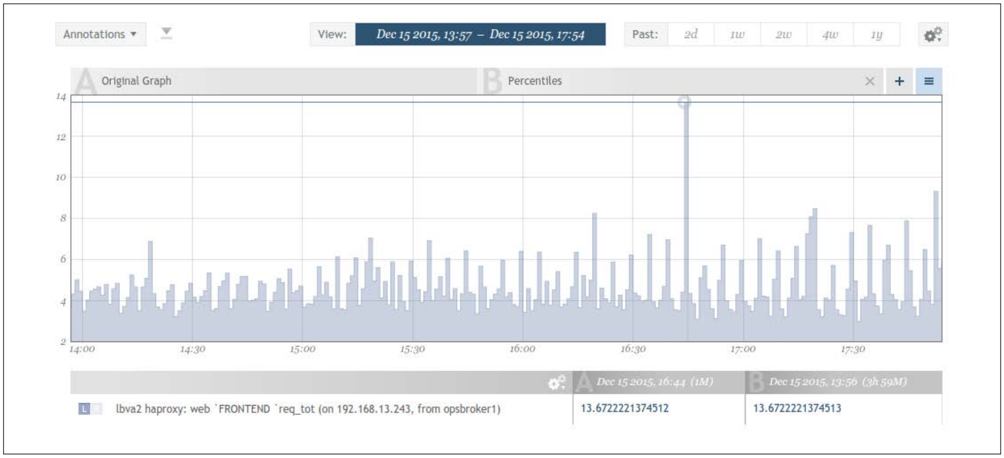


图 2-2：较小时间窗口的平均延时

即便使用完全相同的数据集，图 2-1 中的平均值表明峰值在 7.3 左右，而图 2-2 中的峰值为 14。



警惕存储平均值

记住要存储实际值，而不是平均值。如果一个监控程序每分钟计算平均值，而不保留实际值的完整历史记录，那么你有时需要使用 1 分钟的平均值计算 5 分钟的平均值。你将无法得到正确的数据，因为原来的平均值是有损的。

如果把每分钟的数据看作一个完整的数据集，而不是一个平均数据集，那么可视化异常值的影响会很有价值（事实上，我们可能更关注异常值），这可以通过多种方法实现。可以可视化最小值和最大值与平均值的差异，还可以计算 1 分钟内一定百分比的值的平均值，以剔除异常值 ，比如最快的 99.9%、99% 和 95%。如果对比 3 个值：100% 平均值、最小值和最大值（如图 2-3 所示），就会直观地看到异常值的影响。

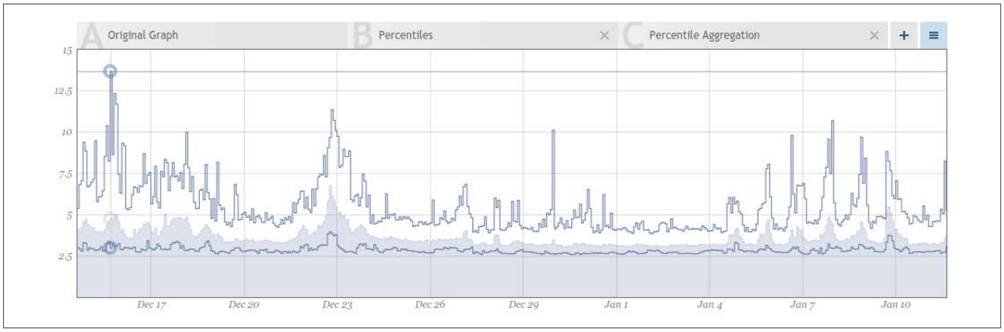


图 2-3：延时平均值（100% 样本量）、最小值和最大值的对比

了解了上述知识后，下面考虑延时 SLO。如果每分钟都取平均值，那么不管 SLO 是什么，都不能证明是否实现了目标，因为只是在测量平均值。为什么不让目标更贴近实际情况呢？可以这样改进 SLO 的定义：在 1 分钟内，对于 99% 的请求，延时必须为 25~100 ms。

为什么选择 99% 而不是 100% 呢？延时往往呈多峰分布。有正常情况也有异常情况，这是由复杂的分布式系统中存在的大量偶然性造成的，比如 JVM（Java 虚拟机）垃圾收集、数据库落盘和缓存失效等。因此，我们预估有一定比例的异常值，设定 SLO 旨在识别可容忍的异常值百分比。

下面考虑负载。我们讨论的是 API 这类简单的响应时间吗？还是测量页面渲染？页面渲染是一段时间内发生的许多调用的综合。如果要测量页面渲染，我们可能需要将初始响应作为第一个需求，将最终渲染作为第二个需求，因为二者的间隔时间可能很长。

2.3.2 可用性指标

如前所述，可用性是服务能够在规定时间内响应请求的时间度量，通常用百分比表示。例如一个系统的可用性是 99.9%，可以表示为：服务必须在 99.9% 的时间内是可用的。

这允许每年有 526 分钟的故障时间，相当于近 9 小时！要求非常宽松。你可能会问，为什么不说 100% ？如果你是产品负责人或销售人员，可能会这样做。人们普遍认为，99%、99.9% 和 99.99% 之间分别存在一个数据量的差异，每一次提升都会让系统变得更复杂、更昂贵，也更容易让工程师分心。另外，如果这是一个通过互联网或远距离传输数据的应用程序，可以预见传输介质将产生影响，导致系统的可用性无法达到 99%~99.9%。

话虽如此，但一年中 526 次一分钟的宕机，与一次 526 分钟的宕机有很大的区别。宕机时间越短，大多数用户越不容易注意到中断。相比之下，如果一些服务中断 8 小时，就会引起新闻报道、数千条推文的讨论，导致用户对其失去信任。围绕服务考虑以下两点是有意义的：MTBF（mean time between failures，平均故障间隔时间）和 MTTR（mean time to recover，平均恢复时间）。通常会优先考虑避免发生故障，这意味着 MTBF 越长越好。MTTR 是发生故障之后恢复服务所需的时间，越短越好。

1. 可用性中的弹性与稳健性

过去十年，有很多关于构建弹性系统的讨论。弹性系统具有如下 3 个特性。

- 由于监控和自动故障修复能力良好，因而 MTTR 较短。
- 由于分布式系统和冗余环境，因而故障的影响范围较小。
- 系统视单机故障为正常场景，并确保自动修复和手动修复的方案拥有良好的文档记录、经过演练且融入日常运维中。

请注意，这里并不关注消除故障。没有故障的系统虽然稳健，但会变得脆弱。当故障发生时，团队很可能没有做好准备，故障的影响范围可能会扩大。此外，可靠但脆弱的系统可

能会导致用户期望的可靠性比 SLO 承诺的更高，而 SLO 正是为此而设计的。这意味着，即使没有违背 SLO，当故障发生时，用户也可能会非常沮丧。

有了这些知识，当评估可用性 SLO 时，应该问自己一些关键问题。

- 在故障期间是否有临时解决方案？能否在降级模式（比如只读模式）下运行？可以使用缓存来提供数据吗（即使数据不是新的）？
- 如果仅局限于一小部分用户，是否有不同程度的容忍度？
- 当故障时间越来越长时，用户体验如何？

  - 一次失败的请求；
  - 30 s；
  - 1 分钟；
  - 5 分钟；
  - 1 小时或更长时间。

基于这些，你可能需要重新评估原来的可用性 SLO，方法如下：

- 定义时间间隔；
- 定义故障最长持续时间；
- 在确定可用性问题之前，定义受影响用户的比例。

之后，可以表达 SLO 如下：

- 一周内的可用性为 99.9%；
- 单次故障不超过 10.08 分钟；
- 如果 5% 以上的用户受影响，则视为宕机。

2. 设计允许的宕机时间

利用新的迭代，可以合理地设计流程，例如故障转移、数据库锁和重启。我们可以进行滚动升级，让受影响的用户不到 1%。如果这周没有发生宕机，并且构建时间少于 10 分钟，则可以用锁表的方式来构建索引。通过为允许的宕机时间进行设计，而不是试图实现零宕机时间，可以提高设计的效率，并且可以为创新和加快迭代速度冒些风险。

值得注意的是，即使在当今世界，99.9% 的可用性也普遍存在，有时候服务仍然可以安全地忍受计划和管理范围内的宕机时间。经过沟通后可以承受 4 个小时的宕机，使用只读选项来减轻宕机影响，并将宕机的影响控制在较小的用户比例上，这样就不必花费几个小时精心安排迁移，因为迁移可能会带来数据损坏、隐私问题等风险。

考虑到这一点之后，你可能希望通过添加规划的宕机来重新评估可用性 SLO，以指导运维团队的工作。

可用性 SLO 样例的迭代版本如下。

- 一周内的可用性为 99.9%。
- 单次故障不超过 10.08 分钟。
- 如果超过 5% 的用户受影响，则视为宕机。
- 允许每年 4 小时的宕机时间，如果：
  - 至少提前两周与用户沟通；
  - 每次影响不超过 10% 的用户。

2.3.3 吞吐量指标

作为服务级别指标，吞吐量应该列出服务必须能够支持的峰值，同时保持与其对应的延时和可用性 SLO。你可能会说："莱恩和夏丽蒂，为什么还需要吞吐量指标呢？难道延时和可用性还不够吗？"我们中的一个会这样回答："无畏的新手，问得好！"然后她会若有所思地吸起烟斗……

有时可能会出现瓶颈，给吞吐量设置了上限，但不至于影响性能或可用性。也许系统中存在锁，限制系统每秒 50 个查询。这种响应可能很快，但是如果有 1000 人等待运行该查询，就有问题了。因为有时无法测量端到端的延时，所以吞吐量指标通常可以作为系统是否满足业务需求的一种额外验证。

当使用平均值和更粗粒度的样本时，吞吐量可能会遇到类似于延时的可见性问题，监控时需要记住这一点。

1. 成本 / 效率指标

当考虑系统成本的有效指标时，最大的变量是采用什么指标来考量成本。这实际上是一个业务决策，但是你应该选择在服务中驱动价值的因素。如果你是在线杂志之类的内容提供者，那么交付的页面至关重要；如果你是软件即服务（SaaS）平台提供商，那么服务订阅指标是有意义的；如果你是零售商，则交易量是合适的指标。

2. 注意事项

数据库工程师为什么需要知道这些？你管理的是服务的一个组件，为何必须关注整体需求？在故障频发的日子里，可能已经为你设定了数据存储目标，并根据你实现该目标的能力进行评级。但是，作为较大团队的一员，你也有机会影响服务的速度和可用性。

通过了解整体 SLO，可以为关注点设置优先级。如果延时 SLO 为 200 ms，那么可以假设这 200 ms 包含以下内容：

- DNS 域名解析；
- 负载均衡；

- 重定向到 http 服务器；
- 应用程序代码；
- 应用程序查询数据库；
- 广域网中 TCP/IP 的传输时间；
- 从存储中读取数据，包括固态硬盘和旋转磁盘（spinning disk）。

因此，如果数据存储做得很好，并且耗时较短，你自然应该关注其他方面。如果发现延时 SLO 效果不佳，并且注意到影响性能的关键点所在，那么可以在 sprint[1] 上花些时间，优化影响较大且容易解决的性能痛点。

在为新服务设定 SLO 时，有一些额外的事情需要考量。

❑ **把握分寸**

我们拥有度量值，理解事情的轻重缓急。但请尽量保持度量值列表简洁，可以在一个页面仪表板上展现 SLO 状态。

❑ **以用户为中心**

想想用户最关注什么。请记住，大多数应用程序服务关注延时、吞吐量和可用性，存储服务还关注数据持久性。

❑ **SLO 是一个迭代过程**

如果有一个 SLO 审查流程，就可以随着时间的推移对其进行修改和添加。早期可能不需要高标准的 SLO，这使得工程师可以专注于功能和改进。

使用 SLO 确定如何设计服务、流程和基础设施。

2.4 SLO的监控和报告

既然已经确立了 SLO，那么相较于理想目标，监控实际运行的表现至关重要。截至目前，还没有开始讨论运维的可见性，但在进入下一个话题前，还有一些关键的事情需要讨论。

监控服务等级管理的首要目标是，及早发现并修复任何潜在的问题，否则这些问题会使我们无法达成 SLO。换言之，我们不想依赖监控告诉我们当下正在违背 SLO。这就如同在皮划艇比赛中不想进入湍流后才发觉，而是想在平静水域时就有征兆表明下游有湍流。然后就可以采取行动，确保不违背对自己和系统许下的承诺。

在监控时，我们将依赖度量值的自动收集和分析。然后将这些数据输入自动化决策软件进行修复，或者用于向操作人员（自己）报警，或者为后续工作创建工单。此外，需要将这些数据可视化以便进行实时分析，可能还需要为当前状态的高级视图创建仪表板。当讨论

注 1：敏捷项目管理方法 Scrum 中的一个概念。—— 译者注

监控的各种指标时，上述情况都要考虑在内。

换言之，假设服务每周只允许宕机 10.08 分钟，到周二时，由于 Cassandra 垃圾收集器发生 STW（stop the world），导致 3 天内 [2] 就已经宕机了 3 分钟，负载均衡故障转移宕机 1 分钟。至此，已经用完了约 40% 的 SLO，而这周还有 4 天。此时就要考虑调整垃圾回收机制了。当超过特定阈值（比如 30%）便报警，并在工单系统中发送电子邮件，以便数据库可靠性工程师直接解决该问题。

2.4.1　可用性监控

下面使用前面定义的可用性 SLO。该如何监控这个指标呢？我们需要监控系统的可用性以及用户级别的错误，并基于此得到合适的监控图。重申一下，所举示例的可用性 SLO 如下。

- 一周内的可用性为 99.9%。
- 单次故障不超过 10.08 分钟。
- 如果超过 5% 的用户受影响，则视为宕机。
- 允许每年 4 小时的宕机时间，如果：
  - 至少提前两周与用户沟通；
  - 每次影响不超过 10% 的用户。

习惯上，运维人员倾向于关注底层的监控，以便了解系统是否可用。例如，他们可能会监测主机是否已启动、是否可达，以及其上的服务是否正在运行并可访问。在分布式系统中，很快就会发现这种做法是不可持续的，也不能很好地预示服务的可用性。假如有 1000 个 JVM、20 个数据库实例和 50 个 Web 服务，我们怎样才能得知，其中是否有某个组件正在影响服务可用性，以及影响程度如何？

关于此，首先要关注用户请求的错误率，也称 RUM（real user monitoring，真实用户监控）。例如，当用户从浏览器提交 HTTP 请求时，他能否收到服务器的正确响应？ 如果服务很受欢迎，可能还会产生大量数据。一个重大的全球性新闻事件，会在 Web 服务上产生每秒 7 万多次的点击。任何现代 CPU 都可以高效地计算此量级数据的错误率。这些数据从应用程序（比如 Apache HTTP）记录到一个日志守护进程（比如 Linux syslog）。

系统从这些日志中获取数据并输出到适当的工具进行监视和分析的方式千差万别。本书暂不讨论这一点，并假设我们已将服务的成功 / 错误数据存储在生产数据库中，而且没有进行任何聚合或平均。前文已有论述，但值得再次强调的是，只存储平均值会丢失有价值的数据。

有了这些数据后，计算每秒的请求失败是否超过 1% 就很简单了。如果超过了，就将这一秒标记为宕机时间。按这种方法计算出总的宕机时间，并与一周所允许的 604.8 s 宕机时间做比较，然后将结果生成报告在仪表盘上展示，以便通过浏览器查看，或在网络操作中心

注 2：西方国家一般将周日作为一周的第一天。—— 编者注

或办公室的监控器上以及任何有助于利益相关者了解团队情况的地方展示。

理想情况下，我们能利用这些数据，预测宕机时间是否会超过当周预算。在大部分环境中，预测面临的最大挑战是由于产品演进而导致的负载转移。在每周甚至每天发布版本的系统中，以前任何数据集的作用都非常有限。与最近的数据集相比，旧数据集尤其如此。这称为**衰减函数**（decaying function）。

本书无意探讨可预测性数据科学，但有很多方法可用于预测当周或者未来几周是否会违背SLO。可以采用宕机时间不超过当前值的前 N 周（在相对稳定的环境中 N 较大，在频繁部署的系统中 N 可能小到 1）的数据，分析这 N 周中违背 SLO 的次数，并基于此来预测。

例如，你的脚本记录了本周当前的宕机时间（10 s）和本周当前的总秒数。该宕机可以定义为：在本周 369 126 s 内，总的宕机时间为 10 s。

然后可以计算前 13 周的数据，对于每个在相同时间点（1~369 126 s）宕机时间不超过 10 s 的周，计算该周是否违背了 SLO。接着基于时间接近程度赋予不同的权重。比如在这 13 周中，给紧邻当前时间的前一周赋予权重 13，再前一周赋予权重 12，以此类推。对违背 SLO 的那些周进行加权求和，如果这个值大于等于 13，就有必要给运维团队提工单，并及时通知他们处理。如果没有训练有素的数据科学家来审查服务等级数据，这是确保有某种级别的数据驱动监控的一种方法。这里的目标是，在潜在问题演变为紧急事件之前发现它，这意味着更少地打扰值班人员和减轻对可用性的影响。

除了 RUM，人工创建测试数据集也是有用的，这称为**合成监控**（synthetic monitoring）。数据集是人工创建的，并不意味着其行为和真实用户不同，就像邮件企业也会像一般用户一样从 QA 账号发邮件。

合成监控可实现持续、完整的覆盖。用户可能来自不同区域，活跃时间也不尽相同。如果不对服务的所有区域和活动都实施监控，就会存在盲点。利用合成监控，可以识别哪些区域的可用性或延时变得不稳定或恶化了，并做好准备或采取缓解措施，例如扩容、性能调优，甚至将流量从不稳定区域迁移走。

有了合成监控和 RUM，就能知道何时可用性受到影响，甚至预测何时可能会违背 SLO。但对于更大的故障，比如系统发生故障或者容量达到上限等情况，这些监控帮助不大。而实施稳健性监控的一个重要原因，就是在发现系统发生故障和过载之前，收集足够多的数据来进行预测。

2.4.2 延时监控

延时监控和请求错误监控非常相似，只不过可用性是布尔值，而延时是时间值，必须测量并验证它是否在目标 SLO 范围内。

延时 SLO

1 分钟内，99% 请求的延时必须为 25~100 ms。

在我们的错误监控中，假设所有 HTTP 请求的日志都已经从 syslog 存入时序数据库中了。有了这些数据，就可以对一定时间间隔内的请求按延时进行排序，并剔除延时最高的 1% 的请求。在本例中，对 1 s 内的请求的延迟取平均值。如果剩余的 99% 的请求中，有任何一个请求的延时超过 100 ms，就算违背了一次 SLO。

有很多工具或脚本可以利用这类数据进行预测性分析。通过测算以往在相似时间内或具有相似流量模式的延时数据，可以寻找表征响应时间变长可能导致违背 SLO 的异常现象。

2.4.3　吞吐量监控

前面收集并检查了可用性和延时 SLO 的数据，有了这些数据，很容易监控吞吐量。如果存储了每条记录，就可以轻松计算出每秒的事务数量。如果该值超过了 SLO 要求的最小事务数量，就表明没问题。如果服务的实际流量达不到 SLO 的吞吐量，则需要定期做负载测试，以确保系统能满足 SLO 的要求。稍后会详细介绍负载测试。

2.4.4　监控成本和效率

成本和效率是很难监控的 SLO，因为有些成本无法量化，为此必须考虑一段时间内的总成本。如果你的服务在云环境中运行，则所用资源的费用已经明确，很容易量化所用资源的成本，例如存储、计算、内存和带宽等。如果使用的是自己的裸机，则需要计算所有用于服务的硬件成本，并估算共享资源的使用成本。此外，成本计算周期并不是细粒度的，所以如果供应商出具的是月度报告，那么了解某个时间段的费用就不太容易，比如按小时计算。

对于成本固定的资源，比如主机和存储资源，可以从供应商或者自己的内部数据库中获取最新数据。在部署或下架资源时，可以参考这类数据来评估成本。对于带宽、IOPS 等用量类型的资源，可以有计划地参考其他已收集的度量值，以评估费用。

也要考虑维护服务的人员成本，包括运维人员、数据库管理员和网络工程师，以及任何待命人员，还包括项目管理人员。这类共享资源，可以按其投入到监控服务中的时间百分比来计算。如果组织正在使用时间追踪工具，那么可以基于这些数据生成实时人力资源的使用情况数据；否则需要做常规的估算，将人员离职、新入职、团队调整等因素都考虑在内。

这些工作都需要人工完成，其中一些无法简单地自动化。尽管如此，这些数据对于评估运维成本是非常有价值的。将该成本和服务产生的价值对比，有助于可靠性工程师在提升效率方面树立目标。

2.5 小结

服务等级管理是架构设计和运维的基石。所有事情都围绕不违背 SLO 这件事开展，这怎么强调都不为过。SLO 确立了行事的基本准则，我们基于 SLO 决定哪些风险是可以承担的、应该选择哪种架构，以及如何设计操作流程以支撑该架构。

本章介绍了服务等级管理的核心概念（包括 SLA、SLO、SLI 等）和常用指标（包括可用性、延时、稳定性和有效性），以及有效监控这些指标以便在违背 SLO 之前发现问题的方法。这将为你提供一个良好的基础，以便有效地沟通对你管理的服务的期望，并为实现这些目标做出贡献。

第 3 章将讨论风险管理，届时将评估影响承诺的服务等级的因素。基于服务等级需求和识别出的潜在风险，可以有效地设计服务架构和操作流程，以确保兑现业务承诺。

第3章

风险管理

运维操作是一系列承诺，以及为兑现这些承诺所要完成的工作。第 2 章讨论了如何确立运维目标，以及如何对该目标进行监控和报告。风险管理就是对会造成违背承诺的不确定性进行识别、评估并划分优先级，也就是利用资源（包括技术、工具、人员、流程）来监控并降低不确定性。

风险管理并不是一门追求完美的科学，它的目标不是消除所有风险，因为消除所有风险不切实际，无疑会浪费资源。风险管理的目标是将评估和降低风险的措施纳入所有流程，并运用缓解和预防技术，通过不断迭代减轻风险造成的影响。这个过程应该是持续的，随着对事故的分析、新架构组件的引入以及组织发展伴随的风险而变化。流程的循环过程可分为以下 7 步。

- 识别可能给服务带来运维风险的危险或威胁。
- 评估每个风险并分析其可能性和影响。
- 对风险的可能性和影响进行分类。
- 确定减轻后果或降低风险可能性的控制措施。
- 评估风险优先级，确定优先处理哪个。
- 实施控制措施并监控其有效性。
- 不断重复上述过程。

重复上述过程，便是实践**持续改善**（Kaizen），也称“持续改进”。该过程在风险评估中尤为重要，因此必须逐步制定策略。

3.1 风险考量因素

影响风险评估流程质量的因素很多，可分为如下几类：

- 未知因素和复杂性；
- 可用资源；
- 人的因素；
- 团队因素。

我们需要考虑所有这些因素，以帮助团队确立一个切实可行的流程，下面简要介绍这些内容。

3.1.1 未知因素和复杂性

当今系统复杂度大幅增加，使得风险评估过程面临更为严峻的挑战。领域知识越复杂难懂，人们将已有知识应用于陌生场景的难度就越大。过度简化概念以便理解被称为**简化偏好**（reductive bias）。这种方法在最初的学习中是有效的，但对于获取更高阶的知识无效。系统中存在大量未知风险，其中很多是我们无法控制的，例如：

- 在托管环境（比如亚马逊或谷歌的云服务）中，来自其他用户的影响；
- 集成到基础设施中的第三方组件的影响；
- 软件工程师提交的代码；
- 促销活动导致的流量激增；
- 上下游服务；
- 补丁、代码库变更，以及其他软件增量变更。

为了评估此类环境中的风险，领域问题的解决有助于评估过程。运维团队必须利用不断积累的经验和知识来构建更完善的规划模型；同时需要认识到，无法考虑到所有可能性，因此必须创建弹性系统，来为未知的可能性做好准备。

3.1.2 可用资源

如果你在资源匮乏的部门或杂乱无章的初创公司工作过，就会知道，为正在进行的或有前瞻性的流程申请资源非常困难。你可能每月只有 4 小时甚至 30 分钟的时间来访问风险管理流程。因此，你必须创造价值。你用于消减风险的时间和资源成本必须少于不作为的成本。换言之，要不断地划分优先级，在有限的时间内优先处理最有可能和会产生最大影响的风险。创建弹性系统并从事故中学习。

3.1.3 人的因素

人们做事情时会有很多潜在问题。人类是聪慧的，但不是完美的。会对风险管理流程产生危害的例子如下。

- **不作为综合征**

  很多运维人员发现，自己的上级或者同事厌恶风险。他们的特点是选择不作为，因为他们认为变更比不作为风险更高。在面对未知情况时，仔细斟酌很重要，但不应退缩不作为。

- **对熟悉的危害视而不见**

  有经验的工程师经常会忽视常见风险，更关注新出现的或者少见的事件。比如经常处理磁盘写满问题的人，可能更关注数据中心的事件，而没有为磁盘空间管理做足准备。

- **恐惧**

  恐惧可以被视为积极的或消极的压力，因人而异。有些人能在高压力、高风险的环境中茁壮成长，并能给计划、风险消减和生产工作创造巨大价值；而有些人则会因为恐惧而忽略最糟糕的情况，这可能导致对关键的高风险的组件和系统缺乏准备。在团队中注意这些反应很重要。

- **过于乐观**

  在风险评估时，另一个倾向是过于乐观。我们经常高估自己以及团队的能力，这会让我们只考虑事物的理想情况（比如不会疲劳，没有其他事故分散精力，可以指挥下属）。这种乐观不止体现在人员评估上，也体现在对事情的评估上。你是否曾认为，“3 块磁盘在同一天发生故障是不可能的”，结果却经历了一批磁盘损坏的窘境？

此外，必须考虑造成风险的身体因素，比如疲劳，并将其视为手动修复（也称“救火”）时的阻碍。任何时候，当我们考虑人工工作及其固有的风险时，比如手动变更和验证，需要假设运维人员刚完成了一整天的工作。可能情况并非如此，但必须考虑到这样的问题。此外，当我们为降低或者消除风险而做风险控制设计时，必须考虑到负责手动修复的人会疲劳，或者可能同时在处理多个问题。

> **报警疲劳**
>
> 报警疲劳（pager fatigue）是由不必要或者大量的报警导致的疲劳感和不堪重负。当决定在监控流程中内置多少报警（需要人工介入和修复的报警）时，应该考虑到这一点。这通常是由于误报（多因阈值设置不合理，对非问题也报警），或者对不久之后可能会有风险的事情采用了报警（alert）而非警告（warning）所造成的。

3.1.4 团队因素

如同个人有盲点，团队也会动态变化，这可能会影响风险管理流程。关于团队因素，需要考虑以下几点。

- **团队极化**

  也称“冒险转移”（risky shift）。团队极化产生的原因是，相比团队中的个人，团队倾向于做出更极端的决定。这种倾向往往使得团队与最初的想法渐行渐远。如果个人观点偏

保守，在达成一致意见后，他们对风险的容忍度会上升；反之，风险容忍会变为风险规避。在团队中，个人往往不想表现得太保守，这种想法会导致团队对风险的容忍度升高。

- ❑ **风险转移**

  当有其他团队分担风险时，团队的风险容忍度往往会提升。比如，我正在为运维团队做计划，如果知道可以向数据库团队求助，我可能会冒更大的风险。树立主人翁意识，并在不能将风险转移给他人的跨职能团队中工作，有助于实现这一目标。

- ❑ **决策转移**

  当团队高估风险时，可能会发生决策转移，这样他们就可以将特定决策的责任转移给别人。如果高风险的变更需要首席技术官批准，并且需要承担责任，人们就会把风险评估得高一些，以便将决策推给他人来做。在依赖团队和个人的专业知识和经验而不是等级审批流程的自治团队中，这种现象会减少。

3.2 可以做什么

现状是，风险管理流程容易变得过于繁重。即使有充足的资源，团队也无法识别影响可用性、性能、稳定性和安全性的所有潜在风险。对于我们来说，创建随时间迭代和改进的流程是有意义的。在处理风险时要力求保持弹性，而不是试图消除所有风险，并允许为了创新和改进而明智地承担风险。

消除所有风险实际上是糟糕的目标。没有压力的系统，不会随着时间的推移得到加固和强化。在面对突如其来的压力时，这些系统往往会变得很脆弱。那些经常面对压力，因而设计得更有弹性的系统，在面对未知风险时能更好地应对。

对于是否应该利用宕机预算（谷歌创造的词），目前尚无定论。谷歌的选择是承担可控范围内的风险，把风险看作获得更多收益的机会。在谷歌，如果每个季度有 30 分钟宕机预算，且这些时间尚未使用，则它愿意承担更大的风险，以换取发布、改进和增强新功能等所带来的收益。这是充分利用预算来创新的做法，而不是完全规避风险。

如何将其转化为现实世界中评估风险的流程呢？首先看看不可以做什么。

3.3 不可以做什么

介绍完需要牢记的要点，下面是深入研究风险管理流程时需要考虑的最后一些建议。

- 不要让主观偏见妨碍流程。
- 不要把偶然事件或者片面言语当作主要的风险评估来源。
- 不要只关注以往的事故和问题，要朝前看。

- 不要止步不前，要不断回顾、总结以往工作。
- 不要忽略人的因素。
- 不要忽略架构和工作流的演进。
- 不要假设环境和以前一样。
- 不要采用脆弱的管控措施，或忽略最坏的情况。

随着时间的推移，你会往列表中添加更多内容，但切记，它们对于思考系统可能遇到的陷阱是有帮助的。

3.4 工作流程：初始版本

无论对于新服务还是已有服务，我们的流程都从初始版本开始。在初始版本（见图 3-1）中，目标是识别出会影响服务 SLO 或很可能发生的主要风险。另外，还应考虑可能危害服务长期可行性的糟糕场景。数据库工程师需要记住，这样做不是为了获知风险的全貌，真正的目标是降低或消除风险，以及计划如何从运维的角度最大限度地利用现有资源。

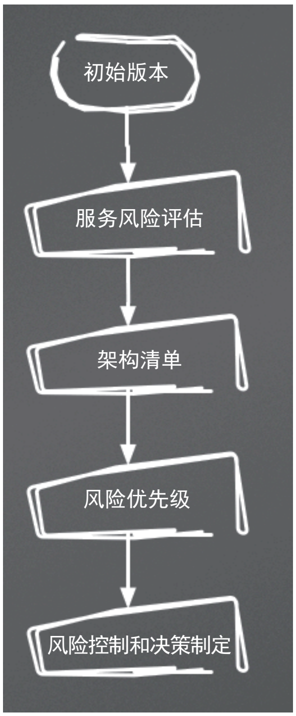


图 3-1：风险管理流程的初始版本

3.4.1 服务风险评估

如果你正在维护一组服务和微服务，那么应该和产品负责人一起评估每项服务的风险容忍度。你应该问以下问题。

- 如何定义服务的可用性和延时 SLO ？
- 当出现下列情况时，宕机或者不可接受的延时是什么样的？
  - 影响了所有用户
  - 影响了部分用户
  - 服务以降级模式（只读模式、关闭某些功能等）运行
  - 服务的性能下降了
- 服务宕机的代价是什么？
  - 收益损失？
  - 用户流失？
    - ♦ 服务是免费还是付费的？
    - ♦ 有用户可以轻易迁移过去的竞品吗？
  - 宕机的影响会危害整个公司吗？
    - ♦ 数据会丢失吗？
    - ♦ 会泄露隐私吗？
    - ♦ 是在举行活动或节假日期间宕机的吗？
    - ♦ 宕机影响扩大了吗？

看一个例子，UberPony 是一家提供骑马服务的公司，涉及以下 6 项服务。

1. 新用户注册。
2. 按需骑马，订单 / 履约。
3. 驯马师注册。
4. 驯马师后勤。
5. 驯马师薪酬。
6. 内部分析。

考虑新用户注册和订单 / 履约两个服务，见表 3-1 和表 3-2。

表3-1：UberPony用户注册服务

可用性 SLO	99.9%
延时 SLO	1 s
每日新增用户	5000
SLO 允许的错误数	5
每天的基础设施成本	13 698 美元
1 美元收入对应的基础设施成本	0.003 美元

（续）

永久会员价值	1000 美元
永久会员带来的每日收入	500 万美元
每分钟用户峰值	100
出错后的用户退出率	60%
每分钟收入损失峰值	6 万美元

表3-2：UberPony订单/履约服务

可用性 SLO	99.9%
延时 SLO	1 s
当前每日订单数	50 万
SLO 允许的错误数	500
每天的基础设施成本	3 万美元
1 美元收入对应的基础设施成本	0.006 美元
每单收益	10 美元
每日收益	500 万美元
每分钟下单峰值	1000
出错后的订单取消率	25%
出错后的用户流失率	1%
每分钟收入损失峰值	2500 美元
每分钟用户价值损失	1 万美元
每分钟总损失	1.25 万美元

表 3-1 和表 3-2 显示，用户注册服务的成本是下单 / 履约服务的 4.8 倍。75% 的用户会重试下单，但是在无法注册时，只有 40% 的顾客会再次光顾。显然，他们愿意尝试 UberDonkey 公司的服务。在本例中，我们尝试考虑了一些因素，比如订单出错后的用户流失率，以及有多少用户或订单在出错之后重试。如果没有商业头脑，很难思考这类问题；但在没有数据可用时，推测也是有用的。

这些数据是会改变的，所以在执行流程时，要确保将数据更新到最新状态。如果 UberDonkey 变得更有竞争力，并且 UberPony 在订单出错后会流失 5% 的用户，那么此时我们订单 / 履约服务宕机的损失就变为每分钟 52 500 美元，成为高优先级的事了。因此，集中精力提升用户注册服务的优先级更为合理。

3.4.2 架构清单

定义完了工作范围，下面为我们负责的系统和环境列出清单：

- 数据中心；
- 架构组件 / 层次（MySQL、Nginx 负载均衡、J2EE 应用实例、网络、防火墙、Hadoop/HDFS、CDN）；

- 组件的角色（写库 / 主库、副本）；
- 服务间的交互和通信方式（从应用到 MySQL 的请求、从负载均衡到应用的请求、从应用到 Redis 的请求）；
- 各种任务（提取、转换和加载，CDN 加载，缓存刷新，配置管理，编排，备份和恢复，日志聚合）。

高优先级服务的精简清单如表 3-3 所示。

表3-3：UberPony用户注册服务

组件	数据中心1统计	数据中心2统计
前端负载均衡	2	2
Web 服务器	20	20
Java 负载均衡	2	2
Java 服务器	10	10
数据库代理	2	2
Cloudfront CDN	服务	服务
Redis 缓存服务	4	4
MySQL 写集群服务器	1	0
MySQL 读集群服务器	2	2
MySQL 复制	服务	服务
CDN 刷新	任务	任务
Redis 缓存刷新	任务	任务
MySQL 备份	任务	不适用
ETL 处理	任务	不适用
RedShift 数据仓库	服务	不适用

下一步是评估该架构中可能影响服务的风险点。

3.4.3 优先级

如何判断那些可能影响服务 SLO 的风险的优先级呢？风险管理领域用导致有害结果的风险的可能性乘以该结果的后果来定义风险。表 3-4 给出了评估范围。

表3-4：风险评估范围

可能性/影响	致命	严重	中等	较小	微小
几乎必然	不可接受	不可接受	高	中等	可接受
非常可能	不可接受	高	高	中等	可接受
有可能	不可接受	高	中等	中等	可接受
不太可能	高	中等	中等	可接受	可接受
几乎不可能	高	中等	可接受	可接受	可接受

为了消除歧义，量化可能性和结果很重要，而结果会随特定领域问题而异。如果对于可能性 / 概率有疑问，可参考 David A. Hillson 的论文“Describing probability: The limitations of natural language”。

下面把可能性的程度分为几个区间，如表 3-5 所示。

表3-5：可能性划分

程度	范围
几乎必然	>50%
非常可能	26%~50%
有可能	11%~25%
不太可能	5%~10%
几乎不可能	<5%

我们将其视为在特定时期（例如一周）内违背 SLO 的百分比。就影响而言，在定义影响的类别和其他可能会摧毁业务的问题时，我们会考虑 SLO。这些问题包括：数据损坏、隐私泄露以及安全事故，这些大都归到“严重”甚至“致命”的类别。同样，这些只是例子。

1. 致命影响（正在违背 SLO）

致命影响的潜在特征如下。

- 整个服务不可用或者降级（延时超过 100 ms），持续 10 分钟或以上，影响了 5% 甚至更多用户（1 周有 10 080 分钟，那么 10 分钟的宕机就违背了 99.9% 可用性的 SLO）。
- 即将或者正在把用户数据暴露给其他用户。
- 让未获授权者访问生产系统或数据。
- 事务数据损坏。

上述任何一项都可能产生致命影响。

2. 严重影响（将要违背 SLO）

严重影响的潜在特征如下。

- 整个服务不可用或者降级（延时超过 100 ms），持续 3~5 分钟，影响了 5% 甚至更多用户（用完了 50% 的可用性预算）。
- 系统容量刚好满足实际需求，不符合留出一倍冗余的目标。

上述任何一项都可能产生严重影响。

3. 中等影响（同一时期发生的其他事故会影响 SLO）

中等影响的潜在特征如下。

- 整个服务不可用或者降级（延时超过 100 ms），持续 1~3 分钟，影响了 5% 甚至更多用户（用完了 33% 的可用性预算）。

- 系统容量还剩 25%，不符合留出一倍冗余的目标。

上述任何一项都可能产生中等影响。

4. 较小影响

较小影响的潜在特征如下。

- 整个服务不可用或者降级（延时超过 100 ms），持续 1 分钟，影响了 5% 甚至更多用户（用完了 10% 的可用性预算）。
- 系统容量还剩 50%，不符合留出一倍冗余的目标。

上述任何一项都可能产生较小影响。

在此提醒，我们不会试图识别所有潜在风险。在日复一日的事故管理和风险管理中，你会添加更多内容。我们现在所做的是**制定框架**，即确定一个有限的范围，以务实的方式约束工作。在本例中，我们基于**最有可能**和**影响最大**的场景来制定框架。

在公有云环境中，比如 UberPony 使用的主机，组件故障和实例故障很常见。换言之，MTBF 很短。因为在任何时候都运行着相当数量的服务器（10 个或 20 个），所以 Web 实例和 Java 实例“很可能”发生故障。话虽如此，Web 实例发生故障意味着有 5% 的用户受影响，Java 实例发生故障意味着有 10% 的用户受影响。这就违背了 SLO，因为可能需要花费 3~5 分钟启动一个新的实例，而这可能会产生严重影响。概率大且影响严重，则必然高风险。如果引入自动修复功能（移除故障实例，并启动一个新实例来替代），经过测试，这个过程平均只需 5 s。这会将影响降至**较小**，因而风险也变成**中等**。

如果架构清单中的组件在服务或者实例级别发生故障，则情况可能会如表 3-6 所示。

表3-6：UberPony用户注册服务

组件	可能性	影响程度	风险
前端负载均衡	有可能	致命	不可接受
Web 服务器	很可能	严重	高
Java 负载均衡	有可能	严重	高
Java 服务器	很可能	严重	高
数据库代理	有可能	严重	高
Cloudfront CDN	几乎不可能	严重	中等
Redis 缓存服务	有可能	严重	中等
MySQL 写服务器	不太可能	致命	高
MySQL 读服务器	有可能	严重	高
MySQL 复制	有可能	严重	高
CDN 刷新	不太可能	较小	可接受
Redis 缓存刷新	不太可能	较小	可接受

（续）

组件	可能性	影响程度	风险
MySQL 备份	不太可能	严重	可接受
ETL 处理	不太可能	较小	可接受
RedShift 数据仓库	几乎不可能	较小	可接受

基于上述框架，首先深入研究任何**不可接受**或**高**风险的事项，然后继续深入研究**中等**风险的案例，以此类推。介绍过运维核心之后会探讨数据库，届时将详细介绍数据库的风险管理。这部分内容旨在帮助你理解相关流程。另一个忠告是，需要考虑整个数据中心层级的风险，尽管这种风险很少发生，但也跟隐私泄露、数据丢失以及可能会导致企业倒闭的其他问题归为一类。

3.4.4　风险控制和决策制定

有了划分优先级的风险列表用于评估，下面了解降低和消除这些风险的控制技术。前面通过在 Web 服务器和 Java 服务器中引入自动替换来缩短了 MTTR。请记住，这里的重点是快速恢复和缩短 MTTR，而不是消除故障。可弹性伸缩的系统好于高可用的脆弱系统。

为何 MTTR 比 MTBF 重要

很少崩溃的系统，本质上是脆弱的。当系统发生故障时，团队是否有把握修复？知道该怎么做吗？经常发生故障的系统有很好的风险控制和缓解措施，以至于可以忽略故障影响，并且团队知道出现问题时应该如何应对。操作流程拥有良好的文档并经过实践检验，自动修复实际上变得非常有用，而不是让问题隐藏在系统的黑暗角落。

对于潜在的风险，团队有以下 3 种选择：

- 规避（找到消除风险的方法）；
- 降低（设法减轻风险发生时的影响）；
- 接受（标记风险是可容忍的，并针对这些风险制定计划）。

从技术上讲，在风险管理领域有第 4 种方法：风险分担——通过外包、保险和其他方法转移风险；然而这不适用于信息技术行业，因此不做讨论。

对于每个组件，我们将考虑故障的类型和影响，以及自动恢复、缩短恢复时间和降低故障频率的控制手段。与之相关联的是成本和工作量，通过比较成本和宕机损失，可以采取合理的措施减轻故障影响。

1. 辨别

在 UberPony 的风险评估中，我们将 MySQL 存储服务的多个层视为高风险，这些是典型的

数据库层。下面看看降低风险的可行措施。我们已经确定了该服务的 4 个关键故障点：

- 写实例故障；
- 读实例故障；
- 复制故障；
- 备份故障。

这些都是数据存储系统的常见故障点。

2. 评估

对于写实例故障，UberPony 运维团队讨论并评估以自动化的方式从 MySQL 写实例故障恢复。如果写实例发生故障，注册服务就不能创建或修改任何数据了。这意味着无法增加新用户，现有用户或 UberPony 也都无法更改用户数据。如果注册服务在高峰期宕机，每分钟损失的用户价值可达 6 万美元。所以，解决这个问题非常重要，**接受风险**的做法不可取。

3. 降低和控制风险

已有一些**消除风险**的方法。我们有一个提供冗余功能的 RAID 10 磁盘系统，因此磁盘故障不会导致数据库故障。在整个环境中也存在类似的冗余。另一种消除风险的方法是用 Galera 替换基本的 MySQL 数据库引擎。Galera 是一种架构，可向 MySQL 集群中的任何节点写入数据。这需要对架构进行重大调整，而且团队中没有人熟悉该引擎。经过考虑，新系统带来的风险似乎超过了所能产生的收益。

如果应用程序设计合理，用户仍然可以登录并从可读实例中查看数据。这就是“降低风险”。在与软件工程团队的交谈中我们得知，这在他们的考虑之中。然而，在降级模式下新用户仍然不能注册，因此该功能不会产生太多收益（在竞争激烈的市场环境中，有些功能未开发，代价是很高昂的）。

最终，团队决定开发自动修复功能，在本例中，就是通过把流量切换到另一个主服务器来自动修复故障。他们选择自动化而不是手动操作，因为 SLO 只允许宕机 10 分钟，手动修复根本来不及。也就是说，在管理写操作时可能会丢失数据，所以该过程必须非常可靠。

4. 实现

团队决定使用 MySQL MHA 实施自动故障转移。MySQL MHA 也称 MySQL 高可用，是一种管理故障转移以及为此变更复制拓扑的软件。在实施这样的关键变更之前，团队拟定了一个稳健的测试计划。测试分阶段进行，从一个没有任何流量的测试环境开始，然后是有模拟流量的测试环境，最后是密切监控的生产系统。多次进行这些测试以确保没有任何异常，测试内容包括：

- 在测试环境中直接关闭主数据库；

- 在测试环境中终止 MySQL 进程；
- 在测试环境中结束运行 MySQL 的服务器实例；
- 模拟网络分区。

每次测试之后，团队会做以下工作：

- 记录故障转移的持续时间；
- 记录模拟环境和生产环境的请求延迟，以评估对性能的影响；
- 验证数据库表是否损坏；
- 验证数据有无丢失；
- 查看来自客户端的错误日志，以了解这段时间的影响。

当系统的表现令团队满意并且符合 SLO 时，可考虑将此过程融入其他日常流程中，以确保它正确执行、拥有清晰的文档并且没有 bug。团队最初选择将其融入部署流程中，通过故障转移流程对数据库对象进行滚动变更，以免影响 MySQL 的单线程复制。当团队完成这些工作后，故障的影响时间缩短到 30 s 甚至更少。

软件工程团队在实施故障转移的过程中，还发现这 30 s 内有可能发生数据丢失。因此，团队让应用程序执行双写，将所有插入、更新和删除发送给事件代理，以便恢复数据。这是进一步缓解写主库故障转移所产生影响的措施。

这只是风险管控的初始版本。重点是记住这并非完美的方案，而只是一系列风险管控手段，只关注优先级最高和价值最大的风险。

随着初始版本的完成，常见风险案例已介绍完毕，下面进入迭代过程。

3.5 持续迭代

有了初始版本，我们将在“架构流水线”（architectural pipeline）中考虑降低和消除故障风险的优先级，进而改进设计、构建和持续的运维操作。之前提到风险管理是一个持续的过程，故开始时不需要考虑全面，因为在持续迭代的过程中会增加风险考量点，以覆盖更大范围，如图 3-2 所示。那么这个流程是怎样的呢？

- 服务交付评审。
- 事故管理。
- 架构流水线。

服务交付评审是对服务演进的定期回顾，着眼于风险容忍度、收入、成本和用户需求的变化。随着这些因素发生显著变化，必须重新考虑以前的风险承受能力以及降低和消除风险的措施，以确保它们仍然是可接受的。

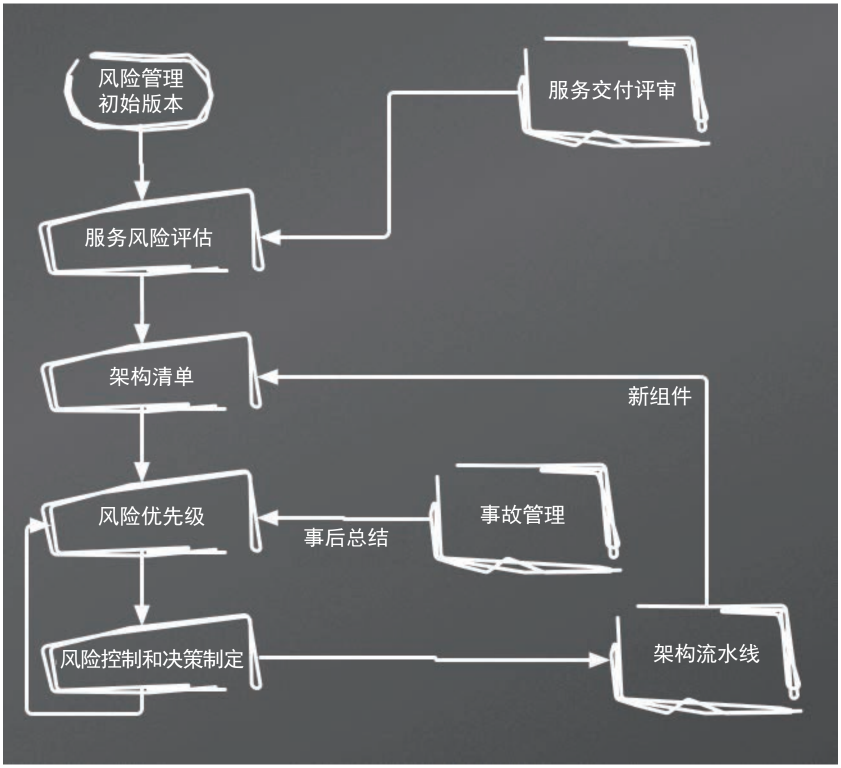


图 3-2：风险管理流程的生命周期及其他考量点

事故管理流程还将引入新的风险考量点。由于事后总结会揭示系统的新脆弱点，因此必须分析这些脆弱点并将它们添加到优先级列表中。最后，随着架构流水线的构建，必须通过风险管理流程引入新组件，以识别设计阶段可能遗漏的任何风险点。

3.6 小结

本章介绍了把风险管理纳入 IT 日常流程的重要性，讨论了可能影响流程的考虑事项和因素，并在日常流程中引入了风险管理的初始版本，以便随着时间的推移逐步完善。

即使了解了服务等级的承诺和这些承诺的潜在风险，仍然缺少一个重要的组成部分：运维可见性。我们需要具备态势感知能力，了解系统过往的性能和特性，以便预见问题，并就如何持续改进系统做出决策。

第 4 章

运维可见性

可见性（通常称为“监控”）是数据库可靠性工程技术的基石。运维可见性（OpViz）意味着，通过定期测量和收集关于各个组件的数据，了解数据库服务的特性。为何它很重要？为何需要运维可见性？部分原因如下。

- **故障 / 修复和报警**
  我们需要知道故障是何时发生的，或者何时会发生故障，以便进行修复，避免违背 SLO。
- **性能和行为分析**
  理解应用程序的延时分布（包括异常值）很重要，我们需要了解实时趋势。这些数据对于了解新功能、实验和优化的影响至关重要。
- **容量规划**
  能够将用户行为和应用程序效率与实际资源（CPU、网络、存储、吞吐量、内存）关联起来，对于确保业务在关键时期不会遇到容量不足的情况至关重要。
- **调试与剖析**
  快速变更意味着服务可能会中断。良好的操作可见性让我们能够快速识别故障点和优化点，以降低未来可能出现的风险。人为错误从来不是根本原因，但系统总是可以改进得更有弹性。

人为错误从来不是根本原因吗

在分析故障或问题时，很容易把错误的根本原因归于人。但是，如果深入研究，就会发现人为错误似乎是由流程或环境的问题导致的。怎么会这样呢？部分原因如下。

- 脆弱、缺乏工具或过于复杂的系统可能会导致人类犯错。
- 流程没有考虑到人的需求，比如休息、环境或技能，也可能导致人类犯错。
- 雇用和培训运维人员的流程可能失效了，导致不合适的运维人员进入了环境。

此外，“根本原因”这样的表述本身就有问题，因为很少有单个问题导致错误和故障的情况。复杂的系统会导致复杂的故障，而人会使得事情变得更加复杂。应当考虑各种因素，并按风险和影响的优先级排列，而不是挖掘根本原因。

- **商业分析**

  了解用户如何使用业务功能是非常关键的指标。普遍而言，了解人们如何使用你提供的功能以及它提供的价值与成本之间的关系是至关重要的。

- **相关性和因果关系**

  通过将基础设施和应用程序中的事件注册到运维可见性系统中，可以快速地将工作负载、行为和可用性方面的变更关联起来，事件的示例包括应用程序部署、基础设施变更和数据库 schema 变更。

组织的各方面都需要真正的运维可见性。本章着重探讨架构的可观测性问题。虽然没有推崇的工具，但是有一些原则、通用分类和使用模式值得了解。下面通过大量案例研究和示例方法来说明。首先考虑运维可见性从传统方法到当前方法的演变。

传统监控

传统的监控系统一般具有以下特点。

- 主机通常是服务器，而不是虚拟实例或容器，它们的生命周期普遍较长，以月为单位，有时以年为单位。
- 网络地址和硬件地址是稳定的。
- 关注点是系统而非服务。
- 对利用率和静态阈值（又名“征兆”）的监控比面向用户的指标（SLI 更多）。
- 不同系统之间有不同的竖井（例如网络和数据库）。
- 粗粒度（间隔 1 分钟或 5 分钟）。
- 关注收集（轮询）和展示，而非分析。
- 管理开销大，而且常常比被监控的服务脆弱。

可以将其归结为一种“传统监控”心态。在传统环境中，数据库管理员主要关注诸如“我的数据库启动了吗”以及“使用是否可持续”这样的问题，而不考虑数据库的行为如何影响用户服务的延时（可查看直方图中的度量值的分布和潜在异常值）。他们也想这样做，但往往缺少相应的工具。

运维可见性非常重要。下面介绍关于如何设计、构建和利用这个关键因素的规则。

4.1 运维可见性的新规则

现代运维可见性假设数据存储是分布式的，而且通常是大规模的。需要认识到，数据的收集甚至数据展示都不如数据分析重要。我们总是希望能快速搞清楚以下两个问题：“对 SLO 有何影响？”“为何工作不正常？”换言之，必须把运维可见性当作商业智能平台来进行设计、构建和维护，而不是将其作为一组实用程序交给运维团队。必须像对待数据仓库或大数据平台一样对待它，毕竟游戏规则已经变了。

4.1.1 把运维可见性视为商业智能系统

在设计商业智能系统时，首先要考虑用户可能提出的问题类型，并基于此进行构建。考虑用户对数据延时（“数据访问速度有多快？”）、数据解析（“用户如何使用数据？”）和数据可用性的需求。换言之，为运维可见性服务定义 SLO（参见第 2 章）。

成熟的运维可见性平台的特点是，不仅可以显示运行应用程序的基础设施的状态，还可以显示在基础设施上运行的应用程序的行为。此外，它还应该能够展示业务的行为，以及业务所依赖的基础设施和应用程序如何影响业务。鉴于此，运维可见性平台必须能够满足运维人员、数据库工程师、软件工程师、商业分析师和公司高管的需求。

4.1.2 分布式易失环境成为趋势

前文提到，随着虚拟化基础设施的使用，数据库实例的生命周期正在变短。尽管它们的生命周期仍比其他组件长得多，但是我们仍然需要收集由短生命周期组件所构成的服务的度量值，而不是单个数据库主机的度量值。

图 4-1 展示了关系型数据存储的一个相当稳定的主 / 副本设置，在这种数据库中一天可能有许多活动。当一天结束时，我们可以看到一个全新的设置，如图 4-2 所示。

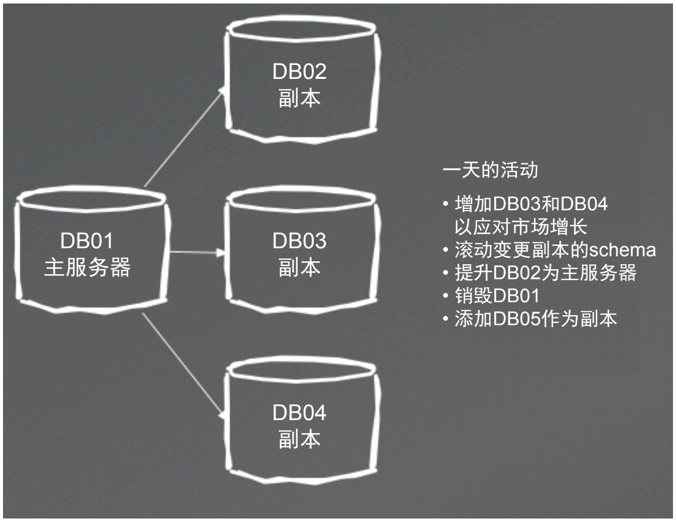


图 4-1：典型的主 / 副本设置

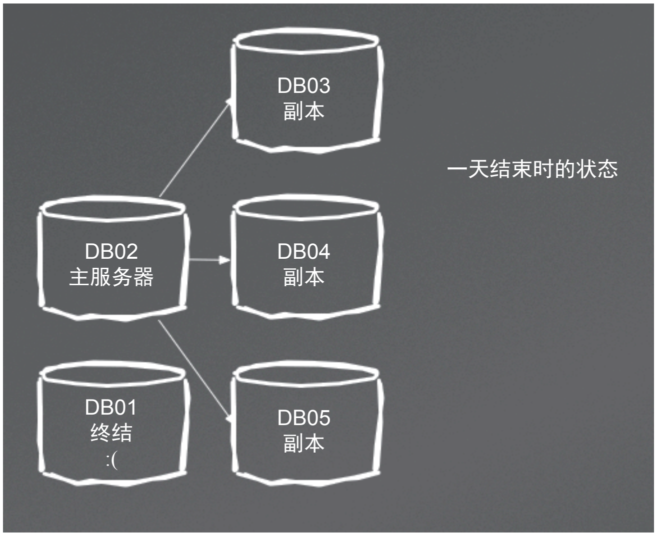


图 4-2：一天结束时

这种动态的基础设施要求我们基于角色来存储度量值，而不是基于主机名或 IP。因此，我们不会将一组度量值存储为 DB01，而是将度量值视为主服务器的，以此观察所有主服务器的行为，即使切换了主服务器。服务发现系统在维护基础设施动态部分之上的抽象方面做得很好，可以促进这一点。

4.1.3 高频存储关键度量值

高频采集度量值对于理解繁忙应用程序的工作负载非常重要。至少，与 SLO 相关的数据都应该保持每秒一次或更高的采样速率，以确保了解系统的运行状况。一条很好的经验法则是，考虑度量值在 1~10 s 内是否有足够的变化来影响 SLO，并基于此确定粒度。

如果你正在监控一个受限的资源，比如 CPU，可能希望以 1 s 或更小的间隔收集数据，因为 CPU 负载变化非常快。对于以毫秒为单位的延时 SLO，数据采集必须足够密集，以检查 CPU 负载是否是造成应用程序延时的原因。数据库连接是另一个例子，采样不频繁的话就可能忽略其影响。

相反，对于不经常变更的条目，比如磁盘空间或服务可用性，可以在不丢失数据的情况下以每分钟一次或更低的采样速率收集测量数据。高频收集数据会消耗大量资源，应明智选择。类似地，采样速率应少于 5 个，以保持运维可见性平台的简单性和结构。

图 4-3 展示了采样间隔太长造成影响的一个例子。

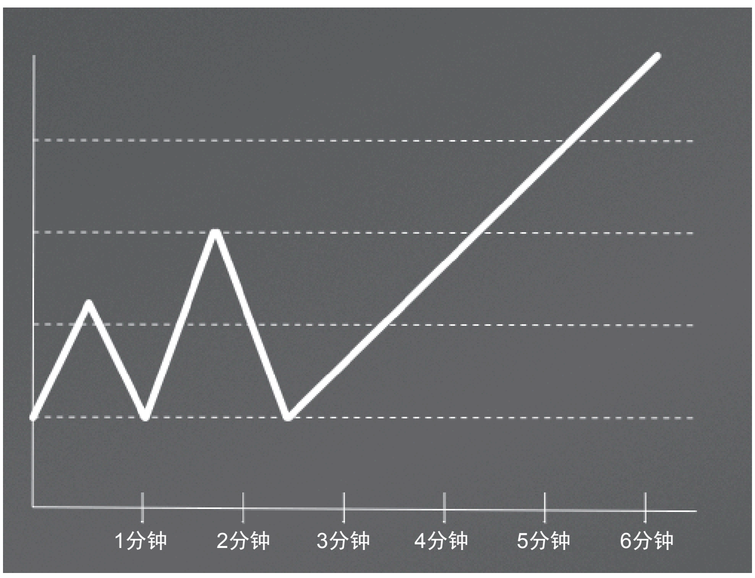


图 4-3：实际工作负载展示了峰值

图 4-3 显示有两个尖峰，随后是长时间的负载增加。现在如果每隔 1 分钟进行采样，图形就会如图 4-4 所示。

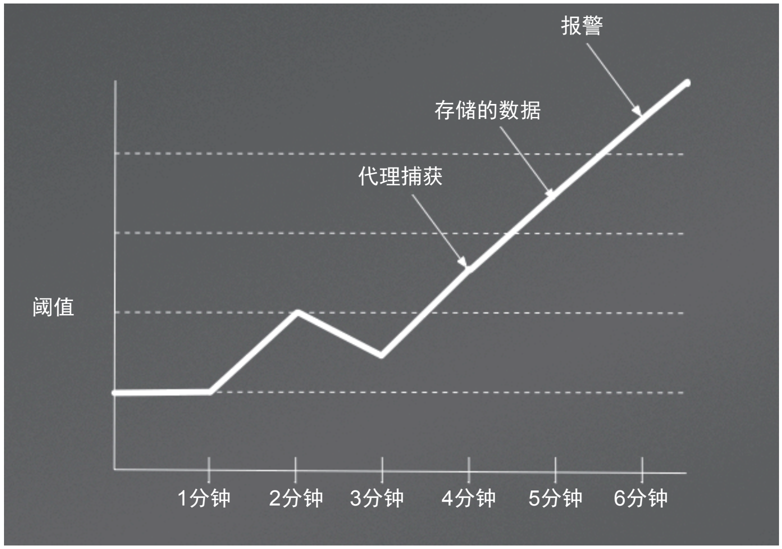


图 4-4：通过每分钟采样可视化工作负载

注意，图 4-4 中甚至没有明显的峰值，线条看起来要平滑得多。事实上，直到第 3 分钟都没有超过报警阈值。假设每分钟存储一次度量值并检查报警规则，我们甚至在违反规则 7.5 分钟之后才会向运维人员报警。

4.1.4 保持架构简洁

对于一个不断增长，并且基础设施中服务的实例 / 服务器随时上下线的系统而言，在不同粒度上检查 1 万甚至更多度量值是很正常的。我们的目标是能够快速地回答上述问题，这就要求不断降低信噪比，而这意味着要严格控制允许进入系统的数据量，特别是在人员交互点，比如展示和报警。

"监控一切"成为监控的口号已有很长一段时间了，它是对监控稀疏且独特的环境的一种反应。事实上，分布式系统和多服务应用程序的度量值过多。处于早期阶段的组织往往没有资金和时间来管理如此多的监控数据，更大型的组织应该有能力关注对系统至关重要的东西。

聚焦度量值

首先关注与 SLO 直接相关的度量值，也称"关键路径"。根据第 2 章所述，运维可见性平台应该优先考虑如下度量值。

- 延时

  客户端调用服务需要多长时间？

- ❑ **可用性**

  有多少调用错误？

- ❑ **调用频率**

  调用服务的频率是多少？

- ❑ **利用率**

  查看服务时，你应该知道如何使用关键资源保证服务质量和容量。

当然，数据库可靠性工程师需要立即开始将这些度量值分解到数据存储子系统中。这是有意义的，而且是一种自然进化，稍后会讨论。

简单性也包括标准化，这意味着标准化模板、解析以及呈现给工程师的任何控制开关和特性。这样做可以确保系统易于理解，从而便于识别和解决问题。

记住这 4 条规则，有助于在设计和构建有价值、有用的监控系统时保持正确的方向。如果发现自己违反了规则，多找找原因。除非确有理由，否则需要考虑最基本的问题。

4.2 运维可见性框架

单是这一话题就够写一本书了。当开始为运维可见性平台收集和准备数据，以便完成工作时，需要能够识别好的平台并提倡采用更好的平台，这就是本节的目标。

可以把运维可见性平台想象成一个巨大的分布式 I/O 设备。数据被传入、路由、结构化，最终从另一端输出，以帮助你更好地理解系统，识别出由损坏或即将损坏的组件所引发的行为，并满足 SLO。详细过程如下。

- 客户端上的代理生成数据，然后发送到该数据类型的中央收集器（用于 Sensu 或 CollectD 的度量值、用于 Graphite 的事件、用于 Logstash 或 Splunk 的日志）：
  - 集中监控系统（比如 Sensu 或 Nagios）除了使用前面提到的 push 方法，偶尔还会使用 pull 方法进行检查；
  - 与必须同时配置代理和监控服务器的 Nagios 等紧耦合的系统相比，分布式检查（应用程序生成检查并转发）的一个优势是，需要的配置管理更少。
- 收集器存储数据（存储到诸如 Graphite、InfluxDB 或 Elasticsearch 之类的系统）或转发给事件路由 / 处理器（比如 Riemann）。
- 事件路由器将数据发送到正确的位置。
- 数据输出包括长期存储、图形、报警、工单和外部调用。

这些都是从运维可见性获得的价值。

4.3 数据输入

要创建输出，需要良好的输入。应尽量使用环境已生成的数据，而不是借助人工探测。通过向系统发送请求来模拟用户被称为**黑盒监控**。**黑盒监控**是从“金丝雀”用户发送，或监控来自互联网边缘的输入和输出。如果流量较小或者运行不够频繁而导致无法有效监控，则黑盒监控是有效的。但是，如果正在生成足够的数据，那么获得真正的度量值（**白盒监控**）将更有价值。**白盒测试**需要对应用程序有深入了解，具体而言，包括检测应用程序的内部逻辑。这方面的优秀工具有 AppDynamics、NewRelic 和 Honeycomb。使用这类工具，可以通过应用程序追踪单个用户流程，一直追踪到数据库。

> **黑盒测试和排队理论**
>
> 鉴于延时的重要性，即使是黑盒测试，也可以利用排队理论、流量信息和调用延时，来确定系统是否饱和。关于排队理论的更多知识，可以参考 VividCortex 的 *The Essential Guide to Queueing Theory* 和新墨西哥大学 Jim Plusquellic 教授的课程“Advanced Computer Architecture”。

该方法的一个优点是，任何创建数据的事物都可以成为代理。随着系统的扩展，生成检查和探测的集中式、单一的解决方案将面临扩展性方面的挑战；但是通过白盒测试，你已经在整个架构中分配了这个任务。这种架构还支持轻松地注册和注销新服务和组件，这得益于运维可见性规则。话虽如此，但有时监控系统可以远程拉取是有价值的，比如检查服务是否可用、监控复制策略是否正在执行，或检查数据库的主机上是否启用了一个重要的参数。

> **从噪声中区分出信号**
>
> 我们越来越依赖更大的数据集来管理分布式系统。现在，我们正在使用大数据系统，管理从应用程序和基础设施中收集的数据。如前所述，利用数据科学和高等数学是当今可观测性领域的明显不足。为了有效地从这些噪声中识别出信号，必须依靠机器从噪声中区分出信号。
>
> 这一领域仍然非常理论化，对于多峰分布的工作负载和由于快速迭代或用户群体变化导致的持续变更而言，大多数异常检测尝试已被证明是不可用的。良好的异常检测系统有助于识别不合规的活动，从而立即指出问题。通过获得更高的信噪比，可以缩短 MTTR。
>
> 以下系统值得考虑：
>
> - Reimann
> - Anomaly.io
> - VividCortex
> - Dynatrace
> - Circonus
> - Prometheus

我们希望把所有有价值的数据发送到运维可见性平台，那么我们说的到底是怎样的数据呢？

4.3.1 遥测/度量值

度量值类型多样，而且几乎无处不在。度量值是对应用程序或基础设施组件的属性的度量。定期观察度量值，创建包含属性、时间戳和值的时间序列。可能适用的属性包括主机、服务或数据中心。这些数据的真正价值体现在通过图表等可视化工具来观察它们时。

度量值通常以以下 4 种方式存储。

- **计数器**

  这些是累积的度量值，表示特定事件发生的次数。

- **仪表盘**

  这些度量值可以在任何方向变化，并指示当前值，比如温度、队列中的作业或活动锁。

- **直方图**

  将许多事件分解成可配置的桶来显示分布。

- **总结**

  类似于直方图，但重点是证明滑动时间窗口的计数。

度量值通常具有相应的数学函数，用于从可视化中获得价值。这些函数创造了更多的价值，但重要的是要记住它们是派生数据，原始数据也很重要。如果你正在追踪每分钟的平均值，但没有原始数据，将无法在较大的窗口上（比如小时或天）创建平均值。以下是一些函数：

- 计数；
- 求和；
- 均值；
- 中位数；
- 百分位数；
- 标准差；
- 变化率；
- 分布。

> **分布的可视化**
>
> 分布的可视化对于查看 Web 架构中生成的数据非常有价值。这类数据很少呈正态分布，并且通常有长尾。在常规图表中很难看出这一点。而采用直方图或火焰图等可视化工具，生成一段时间的分布图，将有助于运维人员了解系统当前的负载状况[1]。

度量值是识别潜在问题表象的线索，因此，对于尽早发现并快速解决可能影响 SLO 的问题至关重要。

4.3.2 事件

事件是在某个环境中发生的离散操作。配置变更、代码部署、主库故障迁移都是事件，通过这些事件可以关联表象和原因。

4.3.3 日志

每个事件都会生成日志，所以可以把日志事件看作事件的子集。对于某些事件而言，操作系统、数据库、应用程序都会生成相应的日志。和度量值不同，日志可以提供额外的信息以及事情发生时的上下文。比如，数据库查询日志可以显示查询操作是何时执行的，以及有关该查询的一些重要度量值，甚至包括执行该查询的数据库用户名。

4.4 数据输出

系统正在接收输入数据，这很不错，但不能直接回答我们的问题也不符合 SLO。运维可见性框架应该输出什么数据？下面进一步研究这个问题。

- **报警**

  报警会打断相关人员的工作，通知他们放下手头工作，去调查导致触发报警的违规行为。该行为的成本很高，应该只在即将违背 SLO 时才采用。

- **工单 / 任务**

  当有工作必须完成时需要创建工单或任务，前提是不会立即发生灾难性故障。工程师工作清单中的工单 / 任务是监控的部分输出。

- **通知**

  有时只是想记录发生过的事件，以帮助创建事件相关信息，例如注册代码部署事件的时间。通常把相应通知记录在聊天室、wiki 或其他协作工具中，而无须打断工作流。

注 1：参见 Brendan Gregg 个人网站中的 Latency Heat Maps 页面。

- **自动化**

  有时数据，特别是利用率数据，会声明需要增加或减少容量。对于这种情况，可以通知自动扩缩容组的同事修改资源池的大小，这是自动化作为监控输出的一个例子。

- **可视化**

  图表是运维可见性最常见的输出，呈现于仪表板以满足不同用户的需求，也是人们进行模式识别的关键工具。

4.5 监控的初始版本

可能你对即将要做的事情感到焦虑，这是正常现象。在此提醒，即将构建的一切只是迭代过程的一部分。从小事做起，让其不断演进，并在需要的时候投入更多精力，在创业环境中尤应如此。

对于全新的创业公司，一切要从零开始。没有度量值，也没有报警和可视化，只有一帮对自己的编码水平高度自信的工程师。许多初创公司在公有云平台创建了作为原型或测试平台的实例，然后将其转换为生产数据库。

你可能刚刚成为一家初创公司的第一位运维人员或数据库工程师，负责管理软件工程师构建的监控和可视化系统，而实际上没有任何监控和可视化系统……

如果你曾供职于初创公司，可能有过这种境遇。不用不好意思，很多初创公司是这样。如果一家初创公司在确定实际需求之前，就构建了一个复杂的操作可视化系统，无疑是一个糟糕的做法。初创公司的成功源于努力打造核心产品，快速迭代，积极寻找用户，快速响应用户反馈和解决生产问题，以及在如何利用宝贵资源上做出艰难抉择。初创公司的成功还源于在需要时及时配备完善的性能可视化系统，而不是提前配备。创业公司失败的原因多种多样，但通常不是因为工程师没能事先预见和度量所有可能的存储指标。最初需要做的只是建立一个**可行的最小可视化监控集**。

> **枚举数据库的动态部分**
>
> 可以把数据看作从客户端到数据库的流。从更高层次看，数据库的职责是接收、保存和返回数据。
>
> - 数据在客户端内存中。
> - 数据在客户端和数据库之间传输。
> - 数据存储到数据库的内存结构中。
> - 数据存储到操作系统和磁盘内存结构中。
> - 数据存储到磁盘上。
> - 数据存储到备份和归档系统中。

对于数据库，需要了解以下方面。

- 获取数据需要多久，为什么？
- 存入数据需要多久，为什么？
- 数据被安全地存储了吗？是如何存储的？
- 主库访问失败时，数据副本可用吗？

当然，这是简化版本，但在深入研究后续内容时，这可以作为一个很好的框架。

在数据库、操作系统、存储系统和各种应用程序层中，可以监控很多度量值。在最基本的需求下，应该能够判断数据库是否宕机；更高层次的目标是监控与真正问题相关的现象，比如连接数和锁的比例。常见的过程如下。

- 监控数据库在运行还是关闭了（pull 检查）。
- 监控总体的延时 / 错误率，以及端到端的“健康检查”（push 检查）。
- 让应用程序层度量每次数据库调用的延时 / 错误率（push 检查）。
- 尽可能多地收集操作系统、存储、数据库和应用程序层的度量值，不论这些数据是否有用。大多数操作系统、服务和数据库有丰富的插件。
- 为已知问题创建具体的检查。比如，当某百分比的节点不在线时进行检查，或者针对过高的全局锁百分比进行检查（主动、迭代地进行这些检查，详见第 3 章）。

有时可以采用第三方监控服务，比如 VividCortex、Circonus、HoneyComb、NewRelic 等，实现高层次的目标。但是，如果把监控数据存储到不同的系统中，则不够优雅，也会让不同监控平台之间的数据分析变得困难。并不是说这样做不好或者不可行，但优雅的技巧能让你走得更远。但在“自我实现”阶段，通常所有监控数据都会使用同一个平台。

现在如果一块磁盘发生故障或者工程师出现失误，需要你来保护公司的业务，你可以问自己一些关于服务健康状况的问题。对于初创公司，几个关键问题是：“我的数据是安全的吗？”“服务运行正常吗？”“用户受影响了吗？”这就是可行的最小监控集合。

4.5.1　数据安全吗

任何业务的关键数据都至少需要 3 个实时备份，即“一主两从”的数据存储，比如 MySQL、MongoDB，或者 Cassandra/Hadoop 之类的三副本分布式存储系统，因为任何时候关键数据都不能只有一个备份。这意味着即使有一个实例发生故障，仍然有一个备份，这就是为什么最小副本数为 3，而不是 2。即使想节约成本，并且每天都在忧心初创公司的发展速度，也不能节省关键数据的备份成本（第 5 章将讨论架构的可用性）。

但并不是所有数据都同等宝贵。如果某些数据丢失造成的损失是可接受的，或者可以从不可变日志中重建数据，那么 n+1 个副本是完全可以的（其中 n 等于服务正常运行所需节点数）。只有知道了每个数据集对公司的重要程度和不可替代性，以及财务的紧张程度，才能做出判断。你仍然需要备份，并且需要定期检查数据备份是否可恢复以及备份过程是否成功。如果没有监控备份是否可以正常工作，就不能认定数据是安全的。

数据安全性监控样例

监控中应包含的一些数据安全性检查如下：

- 3 个数据节点都在线；
- 备份线程正常运行；
- 至少 1 个节点上的备份延迟不到 1 s；
- 最近的备份是成功的；
- 从近期的自动备份进行重建是成功的。

4.5.2 服务运行正常吗

在工具库中，端到端检查是最有效的工具，因为它最能反映用户体验。应该有一个上层的“健康检查”，不仅涵盖 Web 层和应用程序层是否可用，还涵盖所有关键路径上的数据库连接情况。如果数据分到多个主机上存放，则这个检查应该从每个分片上获取一个对象，而且应该自动检查所有分片列表，这样在扩容时就无须手动添加新的检查了。

另外，还有一点很重要，即需要对负载均衡做简单的可用性检查，但它不能消耗过多数据库连接，否则“健康检查”会导致服务不可用。

过度的“健康检查”

夏丽蒂曾经开发过一个系统，其中有一个对 haproxy 做“健康检查”的终端，会在一个 MySQL 表上执行简单的 `SELECT LIMIT 1` 语句。某天他们把一些无状态服务扩容至两倍，执行检查的代理服务器数量也随之翻倍了。在对其他服务扩容时，因“健康检查”导致数据库服务器过载，意外导致整个服务不可用。数据库中 95% 以上的查询都是类似的不合理的“健康检查”，要引以为戒。

教训惨痛，提醒我们应该有额外的监控；如果没有，要对监控服务本身进行“健康检查”。如果数据中心或者云的某个区域宕机，导致其上的整个监控系统不可用，那么不论监控系统本身做得多好或多稳健，都没有意义。所以，为每个关键产品和服务建立外部监控系统，以及对监控服务本身进行“健康检查”是最佳实践。

数据库可用性监控样例

下面是衡量数据库是否可用的几种方法。

- 对应用程序层进行“健康检查”，查询所有前端数据存储。
- 查询每个数据存储系统的每个分区。
- 监控可能出现的容量问题：
  - 磁盘容量；
  - 数据库连接数。
- 抓取错误日志：
  - 数据库重启；
  - 数据损坏。

4.5.3 用户受影响了吗

监测到服务还在工作，这很好。

但是，如果服务的延时是正常情况的两倍或者三倍，又或者 10% 的请求出错了而没有触发“健康检查”，该如何处理呢？如果数据库不可写但依然可读呢？或者副本出现延时，导致大部分写操作挂起呢？如果 RAID 阵列因丢失了一个卷（volume）而以降级模式运行，或者正在建索引，又或是当前出现热点数据，正在写同一行呢？

这正是系统工程特别是数据库工程的有趣之处。系统可能发生各种故障，而我们可能事先只能想到一小部分。

这也是逐渐建立详尽且高层次的服务健康状况度量指标的原因。这些指标包括“健康检查”、错误率、延时等所有对用户造成实质性影响的情况。然后呢？先去干点别的，然后看什么情况下服务会中断。

正如第 3 章所述，事先猜想哪些情况下服务不可用用处不大。既然尚未得到数据，不如去构建其他东西，当服务的某些方面出现异常后再研究。

前面介绍了初始方法和演进方法，下面重点讨论数据库可靠性工程师应该测量哪些数据。

4.6 度量应用程序

首先度量应用程序。尽管大部分问题能从数据存储层度量，但用户或者应用程序行为的变化是首要指标。通过工程师的应用程序度量和应用程序性能管理解决方案（如 New Relic 和 AppPynamics），可以为组织中的每个人获取大量数据。

- 应该已经度量并在日志中记录了某个页面或者 API 的所有请求和响应。
- 应该也对所有对外服务都做了上述度量，对外服务包括数据库、搜索索引和缓存。

- 对于任何任务或者独立的工作流，都应该以相似的方式实施监控。
- 与数据库、缓存和其他数据存储进行交互的可复用代码（例如方法或函数），也应进行类似的度量。
- 监控每个终端、页面或者函数 / 方法调用数据库的次数。

追踪每个操作的数据访问代码（比如 SQL 调用），以便快速地和数据库中详细的查询日志相互参考；但对于 ORM（object-relational mapping，对象关系映射）而言这是一个挑战，因为该系统中 SQL 是动态生成的。

SQL 注释

当对 SQL 调优时，将数据库中运行的 SQL 语句和调用该语句的代码位置对应起来很困难。许多数据库引擎可以添加注释，这些注释会在数据库查询日志中打印出来，这是记录代码位置的好地方。

4.6.1 分布式追踪

追踪从应用程序到数据存储层每个阶段的性能，对于解决很难捕捉到的长尾延时问题非常重要。像 New Relic 或 Zipkin 等开源解决方案，可以实现从应用程序调用到外部服务（比如数据库）的分布式追踪。理想情况下，从应用程序到数据存储的完整事务追踪，应该掌握每个外部服务调用的耗时，而不仅仅是数据库查询的耗时。

数据库的追踪完全可视化之后，将成为培训软件工程师以及培养独立精神的有力工具。你不需要告诉他们关键点在哪儿，他们可以自己获取该信息。就如同 Last Pickle 公司的 Aaron Morton 在演讲中谈到的："用 Zipkin 替代 Cassandra 的追踪功能。"

> 事先预测哪些工具会带来这种积极的文化转变基本上是不可能的，但是我从 Git 及其 pull request 实践，以及稳定的 master 分支中看到了这种转变，也从 Grafana、Kibana 以及 Zipkin 中看到了这种转变。

更多论述，可以参阅 Last Pickle 的博客。

很多组件会进行端到端的调用。数据库可靠性工程师需要了解的组件包括但不限于以下几类：

- 和数据库或者数据库代理服务器建立连接；
- 将一个连接加入数据库连接池队列中；
- 将度量值或事件加入消息或队列服务中；
- 通过集中式 UUID 服务创建用户 ID；
- 基于某个变量（比如用户 ID）选择分片；
- 在缓存层进行查找、校验和缓存；

- 在应用程序层进行压缩或加密；
- 在搜索层进行查询。

> **传统 SQL 分析**
>
> 从事咨询工作的时候，我（莱恩）经常碰到服务没有监控的情形，既没有应用程序层性能监控，也没有数据库监控。我总是要调查 TCP 或收集 SQL 的日志，以便创建数据库的视图。然后，我会列出需要优化的 SQL 优先级列表，并反馈给软件工程师，他们可能并不知道问题代码在哪儿。寻找问题代码可能会花费一周甚至更长时间。
>
> 数据库可靠性工程师有很多机会和软件工程师一起工作，并确保所有类、方法、函数以及任务都和调用的 SQL 做了映射。当软件工程师和数据库可靠性工程师使用相同的工具时，数据库可靠性工程师可以指出问题的关键之处，很快软件工程师就能独当一面了。

如果一个事务有性能“预算”，并且已知延时需求，那么建议各组件的负责人以团队合作的方式开展工作，以便找到相应关键点，并通过适当的投入和妥协来达到目的。

4.6.2 事件与日志

应该收集并保存应用程序的所有日志，包括栈追踪，这是不言自明的。此外，还有许多非常有用的事件可以发送到运维可见性平台，例如：

- 部署代码；
- 部署时间；
- 部署过程中出现的错误。

应用程序监控是至关重要的第一步，从用户的角度观察行为，并且和延时 SLO 直接相关。这提供了服务故障或者降级的线索。下面介绍有助于分析根本原因和建立预防措施的支撑数据：主机数据。

4.7 度量服务器或实例

数据库实例在独立主机中运行，主机可以是物理机或虚拟机。可以从主机获取关于操作系统和数据库运行时分配的物理资源的所有数据。尽管这些数据和具体的应用程序 / 服务不直接相关，但在分析应用程序层延时或者错误时，这些数据是有用的。

当使用这些数据分析应用程序异常时，目的是发现资源被过度使用或未充分使用、资源饱和度以及错误等（即 Brendan Gregg 所定义的 USE 方法）。这些数据对于预估容量增长和性能优化都很重要。认清瓶颈和约束条件，有利于对优化工作排列优先级以产出最大价值。

分布式系统聚合

请记住，单个主机的数据只用于判断该主机是否健康，如果表明主机有问题，则需要从集群中将其移除。相反，要从执行相同功能的主机池的角度考虑资源使用率、资源饱和度以及错误。换言之，如果有 20 台 Cassandra 主机，那么更应关注资源池的总体使用情况、当前的等待时间以及错误情况。如果只有某台主机出错，那么需要从集群中移除它并用新的主机替代。

对 Linux 操作系统而言，需要监控的资源包括如下几项：

- CPU；
- 内存；
- 网络接口；
- 存储的 I/O；
- 存储容量；
- 存储控制器；
- 网络控制器；
- CPU 互联通道；
- 内存互联通道；
- 存储互联通道。

了解所用的操作系统

关于深入研究操作系统特性的重要性，再怎么强调都不为过。尽管很多数据库专家将其视为系统管理员的职责，但数据库服务等级和操作系统之间的联系太紧密了，不得不深入研究。关于这一点，一个很好的例子是，Linux 会将内存都用于 Page Cache，所以对于监控内存使用而言“可用内存”是无用的，而每秒执行的 Pagescan 数量更有意义。如果对 Linux 的内存管理原理了解不深，这一点并不是显而易见的。

除了监控硬件资源，操作系统还有一些点需要追踪：

- 内核互斥量；
- 用户互斥量；
- 任务数；
- 文件描述符数量。

如果不了解这些概念，建议阅读 Brendan Gregg 的博客文章“The USE Method”。这篇文章详细介绍了如何监控这些数据。他为了探究这些数据付出了大量的时间和精力。

事件和日志

不止度量值，应该把所有日志发送到合适的事件处理系统（例如 RSyslog 或 Logstash）中。这些日志包括内核、定时任务（cron）、身份验证、邮件和普通消息日志，以及特定于进程或应用程序（例如 MySQL 或 nginx）的日志。

配置管理系统和资源部署流程都应该向运维可见性平台注册关键事件，可以从如下事件着手：

- 下线某个主机；
- 配置变更；
- 主机重启；
- 服务重启；
- 主机崩溃；
- 服务崩溃。

> **云和虚拟化系统**
>
> 对于这些环境，有一些额外的考量项。
>
> 成本！使用这些环境是按需付费的，而不是像数据中心环境那样预付费。要选择划算的方式。性价比很重要。
>
> 监控 CPU 时，需要监控“steal time”，这是虚拟 CPU 等待实体 CPU 的时间，其间实体 CPU 用于他处。较高的 steal time（10% 或者持续一段时间）意味着环境中有“吵闹的邻居”。如果所有主机上的 steal time 值都较大，很可能是你自己的问题，你可能需要扩容或重新平衡。
>
> 如果只有一台或者少数主机上的 steal time 较高，这意味着其他租户正在占用资源。此时最好停掉这台虚拟机，另启一台虚拟机进行替换。将新虚拟机部署在其他地方可以获得更好的性能。

把这些集成到运维可见性平台，有助于理解主机和操作系统层面正在发生什么。下面研究数据库本身。

4.8 度量数据存储

对于数据库，需要监控和追踪什么？为什么？这些问题取决于具体的数据存储种类。下面重点探讨通用领域，但是会足够详细，以便帮助你追踪自己的系统。主要关注如下方面：

- 数据存储连接层；
- 内部数据库可见性；

- 数据库对象；
- 数据库调用 / 查询。

下面依次介绍各个方面，首先是数据存储连接层。

4.9 数据存储连接层

前面讨论了追踪整个事务中与后端数据存储建立连接所用时间的重要性。追踪系统也应该区分到代理花费的时间与从代理到后端花费的时间。当 Zipkin 这类工具不可用时，可以使用 tcpdump 和 Tshark/Wireshark 临时采样。可以进行自动化采样，或者手动临时采样。

如果应用程序和数据库之间的连接有延时或出错，就需要额外的度量值来帮助找出原因。推荐使用前面提到的 USE 方法。下面介绍其他有用的度量值。

4.9.1 利用率

数据库支持的连接数是有限的，很多地方可能限制数据库的最大连接数。数据库配置参数直接限制数据库可接受的实际连接数，设置上限旨在避免主机过载。追踪连接数上限和实际使用的连接数非常重要，因为上限可能是偏低的默认值。

连接也会用到操作系统级别的资源，例如 PostgreSQL 为每个连接启动一个 UNIX 进程，MySQL、Cassandra 和 MongoDB 为每个连接使用一个线程。它们还会用到内存和文件描述符。理解连接的行为需要考虑很多方面。

- 连接数上限和连接计数。
- 连接的状态（工作中、睡眠中、取消等）。
- 内核级别最大文件句柄的利用率。
- 内核级别最大进程数的利用率。
- 内存利用率。
- 线程池相关度量值，例如 MySQL 表缓存或 MongoDB 的线程池利用率。
- 网络吞吐量的利用率。

这些数据可以表明连接层的容量或利用率是否存在瓶颈。如果利用率是 100%，并且饱和度也很高，就是一个很好的指标；但是，利用率和饱和度都较低也可能表示某处有瓶颈。资源利用率较高但是未完全利用通常也会影响乃至导致延时。

4.9.2 饱和度

饱和度和利用率搭配使用通常更有用。当出现大量等待资源的情况且资源利用率为 100% 时，就表明容量明显有问题。然而，当出现等待 / 饱和但资源未充分利用时，可能别处出

现了瓶颈。通过度量以下值可以得到饱和度：

- TCP 连接的 backlog；
- 数据库连接队列，例如 MySQL 的 back_log；
- 连接超时错误；
- 等待连接池中的线程；
- 内存交换；
- 加锁的数据库进程。

队列长度和等待超时对于理解饱和度非常重要。连接或者进程出现等待是存在潜在瓶颈的征兆。

4.9.3 错误

借助利用率和饱和度，可以判断容量限制和瓶颈是否正在影响数据库连接层的延时。这对于判断是否需要增加资源、删除人为配置的限制或调整架构，是非常重要的信息。应该监控错误，识别并消除故障或配置问题。可以通过以下方式捕获错误。

- 当数据库层发生故障时，数据库日志会提供错误码。有时会配置不同的日志级别，以确保能够发现连接错误。但是需要注意，日志不宜过多，尤其在日志跟数据库共享存储和 I/O 资源时。
- 应用程序和代理的日志也可以提供丰富的错误信息。
- 应该利用前文提到的主机错误。

错误包括网络错误、连接超时、身份验证错误和连接终止等。这些信息可以指明各种错误：表损坏、依赖 DNS、死锁、验证变更等。

利用应用程序的延时 / 错误度量值，追踪并适当地遥测利用率、饱和度和特定错误状态，这些信息足以用来识别数据库连接层的降级和中断状态。下面考虑连接内部应该度量什么指标。

排查 PostgreSQL 的连接速度问题

Instagram 是选用 PostgreSQL 作为关系型数据库的公司之一。它选用连接池组件 PGBouncer 来增加数据库的应用程序连接数。这是一种经过验证的可扩展的、增加数据存储连接数的方法。对于每个连接，PostgreSQL 都会创建一个 UNIX 进程，所以新建连接较慢且开销较大。

Instagram 使用的是 psycopg2（Python 驱动），之前 `autocommit` 参数采用默认值 FALSE，这意味着即使是只读查询，也会发送 BEGIN 和 COMMIT 指令。在把 `autocommit` 改成 TRUE 后，查询延时减少了，连接池中排队等待的现象也有所缓解。

当连接池利用率达到 100% 时，应用程序的延时会增加，最终导致等待连接的请求增加。查看连接层的度量值和 PGBouncer 的连接池监控，就会发现**等待**连接池现象的增加是因为饱和，并且大多数时间**活跃的**连接满负荷。如果没有其他度量值清楚地表明利用率 / 饱和度过高和出错，就应该查看连接内部发生了什么导致查询变慢，稍后会讨论。

4.10 数据库内部可见性

查看数据库内部，就会发现动态部分、度量值的数量和整体复杂性都大大增加了，更接近现实了。重申一下，要牢记 USE 方法。我们的目标是理解可能会影响延时、限制请求或导致错误的瓶颈。

重要的是能从单机视角考虑这些问题，并按照角色聚合。有些数据库，例如 MySQL、PostgreSQL、Elasticsearch 以及 MongoDB，有主 / 副本角色之分。Cassandra 和 Riak 没有特别划分角色，但是它们通常按照 region 或者 zone 分布，按照这些维度聚合也很重要。

4.10.1 吞吐量和延时度量值

数据存储中正进行着多少个和多少种操作？这些数据是数据库活动的一个非常好的高层视图。当软件工程师添加新功能时，这些工作负载将发生变化，并表明负载如何变化。为了解这些不断变化的工作负载而收集的一些度量值如下。

- 读。
- 写：
  - 插入；
  - 更新；
  - 删除。
- 其他：
  - 提交；
  - 回滚；
  - DDL 语句；
  - 其他管理任务。

这里讨论的延时仅指聚合的平均延时。后文会进一步讨论更细粒度、更详细的查询监控。因此，从这些数据中去掉异常值之后，留下来的仅仅是基本负载信息。

4.10.2 提交、重做和日志

尽管具体的实现依赖特定数据存储，但几乎所有落盘都包含一系列 I/O 操作。在 MySQL 的 InnoDB 存储引擎和 PostgreSQL 中，写操作在缓冲池（内存）中被改变，操作被记录在

重做日志中（在 PostgreSQL 中是预写日志）。在维护检查点（恢复时需要用到）时，这些数据由后台进程负责刷新到磁盘。在 Cassandra 中，数据存储在 memtable（内存）中，但是会附加一条提交日志。memtable 会周期性地刷新到 SSTable（sorted-string table，排序字符串表）格式的文件。此外，会周期性地整理 SSTable 文件。可能需要监控的度量值包括：

- 脏缓冲（MySQL）；
- 检查点年龄（MySQL）；
- 正在等待和已经完成的整理任务（Cassandra）；
- 被追踪的脏字节（MongoDB）；
- 修改页和未修改页的淘汰（MongoDB）；
- `log_checkpoints` 参数配置（PostgreSQL）；
- `pg_stat_bgwriter` 视图（PostgreSQL）。

所有设置检查点、落盘和整理的操作都会对数据库活动的性能产生重大影响。有时这个影响会导致 I/O 压力变大，有时在执行重大操作时可能停止所有写操作。收集此类度量值可以调优特定配置，最大限度减轻这些操作发生时带来的影响。因此，在这种情况下，当发现延时增加并且落盘相关度量值显示后台活动过度时，就意味着要调优这些点了。

4.10.3　复制状态

在多个节点间复制数据，可以使一个节点上的数据与其他节点上的完全相同。复制是提升可用性和读性能的基石，也是从灾难中恢复和保证数据安全的一部分。复制共有 3 种状态，然而如果不抓取相应数据并监控，它们可能处于不健康的状态，并且会导致更大的问题。第 10 章将详细讨论复制。

第 1 种故障状态是复制延迟。有时将修改应用于其他节点可能会导致运行变慢。这可能是由网络饱和、单线程来不及 apply 以及其他许多原因导致的。有时，在活动高峰期，会因复制落后导致副本上的数据落后数小时。这很危险，因为提供的可能是旧数据。如果将该副本用作故障转移，可能会丢失数据。

大多数数据库系统提供可以轻松追踪复制延迟的度量值，显示主服务器上的时间戳和副本服务器上的时间戳之间的差异。在类似于 Cassandra 这种使用**最终一致性模型**的系统中，当某些副本不可用后，应该检查用于同步的操作是否堆积，这种机制在 Cassandra 中叫 Hinted Handoff。

第 2 种故障状态是复制机制失灵。在这种情况下，负责数据复制的进程因为某些错误而停止工作。解决该问题需要合适的监控以促进快速响应和修复错误，并且允许继续复制和跟上进度。在这种情况下，可以监控复制线程的状态。

最后一种故障状态最难处理：复制漂移（replication drift）。在这种情况下，数据没有同步，

导致复制过程不可用并有潜在的危险。识别大规模数据集的复制漂移颇具挑战性，并且与负载和存储的数据类型有关。

如果数据相对不变并且写入 / 读取操作是标准的，那么可以在副本间进行校验和检查，比较它们是否一致。可以通过滚动的方式在复制之后进行检查，以便在 CPU 相对空闲的数据库主机上进行安全检查。然而，如果有大量修改操作，将更具挑战性，只能反复进行数据校验和检查（其中有的数据可能已经检查过了）或者偶尔进行采样。

4.10.4　内存结构

数据存储在常规操作中维护着很多内存结构。在数据库中最常见的是数据缓存，尽管它可能有很多名称，但目标都是将频繁访问的数据保存在内存中，而不是磁盘上。其他类似的缓存有：已解析的 SQL 缓存、连接缓存、查询结果缓存等。

监控这些结构的典型度量值如下。

- **利用率**
  在时间维度上分配给缓存使用的总空间。
- **搅动**
  需要缓存其他对象，或者因为底层数据失效导致缓存对象被删除的频率。
- **命中率**
  使用缓存数据的频率，这有助于提升性能。
- **并发**
  这些结构有各自的串形化方法，比如互斥量，这可能会成为瓶颈。了解这些组件的饱和度也有助于优化。

有些系统（比如 Cassandra）使用 JVM 管理内存，监控整个内存的新区域。在这样的环境中，垃圾回收和堆空间的使用也非常重要。

4.10.5　锁与并发

关系型数据库通常使用锁来协调会话间的并发访问。在进行读写操作时，加锁能避免其他进程修改相应的数据。尽管这非常有用，但进程等待会导致延时。某些情况下，进程会因死锁而超时，除了回滚，对于已存在的锁根本没有解决办法。第 11 章会详细讨论锁实现。

监控锁包括监控在数据存储中等待锁的时间。可以将其看作关于饱和度的度量值。较长的队列意味着存在应用程序和并发性问题或者底层出现了影响延时的问题，持有锁的会话需要更长的时间才能完成。监控回滚和死锁同样非常重要，因为这是应用程序没有将锁释放

干净而导致等待中的会话发生超时和回滚的另一个指标。回滚是一种正常的事务行为，但是它们通常是某些底层事件影响了事务的重要指标。

如前所述，数据库中有很多元素作为同步原语，用于安全地管理并发，它们通常是互斥量或者信号量。互斥锁是一种同步对资源（如缓存项）的访问的加锁机制。只有一个任务可以获得互斥量，这意味着互斥量有“所有权”的概念，并且只有锁的持有者才可以释放锁（互斥锁）。这可以防止数据损坏。

信号量限制同一时刻某资源的使用者数量。线程可以请求访问资源（减少信号量），并且可以发出信号，表明资源使用完毕（增加信号量）。使用互斥量 / 信号量监控 MySQL 的 InnoDB 存储引擎的实例如表 4-1 所示。

表4-1：InnoDB信号量活动度量值

名　　称	描　　述
互斥量操作系统等待（增量）	InnoDB 中信号量 / 互斥量等待操作系统返回的数量
互斥量轮次（增量）	内部同步数组中 InnoDB 中信号量 / 互斥量自旋的轮次
互斥量自旋等待（增量）	内部同步数组中 InnoDB 中信号量 / 互斥量自旋等待的数量
操作系统预留统计（增量）	InnoDB 中信号量 / 互斥量的等待添加到内部同步数组
操作系统通知统计（增量）	内部同步数组中 InnoDB 中信号量 / 互斥量的通知统计
排他锁操作系统等待（增量）	InnoDB 中排他信号量（写）等待操作系统返回的数量
排他锁自旋的轮次（增量）	InnoDB 同步数组中排他信号量（写）自旋的轮次
排他锁自旋（增量）	InnoDB 同步数组中处于等待状态的排他自旋信号量（写）数量
共享锁操作系统等待（增量）	InnoDB 中共享信号量（读）等待操作系统返回的数量
共享锁自旋的轮次（增量）	InnoDB 同步数组中共享信号量（读）自旋的轮次
共享锁自旋（增量）	InnoDB 同步数组中处于等待状态的共享自旋信号量（读）数量
每次互斥量等待中自旋（增量）	InnoDB 信号量 / 互斥量自旋轮次跟互斥量等待内部同步数组而自旋的比例
每次排他锁等待中自旋（增量）	InnoDB 排他信号量 / 互斥量（写）自旋轮次跟互斥量等待内部同步数组而自旋的比例
每次共享锁等待中自旋（增量）	InnoDB 共享信号量 / 互斥量（读）自旋轮次跟互斥量等待内部同步数组而自旋的比例

这些值的增长表明并发访问数据存储特定区域的代码达到并发阈值。可以通过配置调优来解决这个问题，或者通过水平扩展使数据存储足以应对持续的并发请求，以便满足业务流量的需求。

当到达扩展的临界点时，锁和并发甚至会使性能较高的查询变得很慢。在生产环境和负载测试中，追踪和监控这些度量值可以了解当前数据库软件的上限，以此确定如何优化应用程序，以应对来自用户的大量并发请求。

4.11　数据库对象

了解所使用的数据库及其存储数据的方式非常重要。简单来说，就是了解每个数据库对象及其关联的键 / 索引占用存储空间的大小。就像对于文件系统存储一样，了解增长率以及何时达到上限与了解存储的当前使用情况同样重要。

除了了解存储和增长，监控关键数据的分布也有帮助。例如了解上界、下界、平均值以及数据的基数有助于了解索引和扫描性能。这对于整数数据类型和低基数字符数据类型特别重要。软件工程师可以利用这些数据优化数据类型和索引。

如果已使用键范围或列表对数据集进行了分片，了解数据分布状况有助于确保每个节点上的输出最大化。这些分片方法可能会出现热点，因为它们甚至没有使用散列或取模方法来分布数据。认识到这一点后，可以考虑重新平衡或替换分片模型。

4.12　数据库查询

取决于所用数据库，实际的数据访问和操作方式可能是高度仪表化的，也可能不是。在繁忙的系统中查询大量数据（这些查询会记录过多日志），可能会严重影响系统的延时和可用性。不过，这些日志是很有价值的。某些解决方案（例如 VividCortex 和 Circonus）专注于从 TCP 和其他协议获取所需数据，这极大地减轻了记录查询日志对性能的影响。其他方法包括：在负载较少的副本上采样、仅在指定时间段内打开日志和仅记录执行缓慢的语句等。

无论如何，我们都希望尽可能多地记录数据库活动的性能和利用率数据。这包括 CPU 的消耗和 I/O，读取或写入的行数，详细的执行时间和等待时间，以及执行计数。了解优化器路径、使用的索引，以及关于连接、排序和聚合的统计信息对于优化也很关键。

4.13　数据库断言和事件

数据库和客户端的日志包含丰富的信息，尤其是断言和错误。这些日志能提供重要的数据，而且这些数据无法通过其他方式进行监控，例如下面这些：

- 尝试连接和连接失败；
- 崩溃时的警告和错误信息；
- 数据库重启；
- 配置变更；
- 死锁；
- 核心转储和栈追踪。

可以聚合其中部分数据，并将其推送到度量系统中。还应该追踪其他事件并用于事件关联。

4.14 小结

本章深入探讨了运维可见性的重要性，并介绍了如何开始构建运维可见性平台，以及如何构建和演进运维可见性架构。对于正在构建和运行的系统，你永远得不到足够的信息。很快你就会发现构建观测这些服务的系统是运维职责的一部分。它们类似于其他基础设施组件，需要得到充分关注。

第 5 章

基础设施工程

本章开始实现应用程序和分析人员所用的数据库集群。前面讨论了大量准备工作：SLO、风险分析以及运维可见性。接下来的两章将讨论设计和构建对应环境的技术和模式。

本章主要讨论数据存储依赖的各种主机环境，包括 Serverless 和 DaaS（database as a service，数据库即服务）。我们将讨论这些数据存储可用的各种存储选项。

5.1　主机

如前所述，数据存储不能存在于真空环境中，它们是在主机上运行的进程。数据库的主机通常是物理服务器。过去 10 年中，选项增加了，出现了虚拟机、容器以及 Serverless。下面讨论每个选项的优缺点，并研究一些具体的实现细节。

5.1.1　物理服务器

这里的物理服务器是指装有操作系统的独立主机，专门用于运行特定服务。在不成熟的环境中，物理服务器在流量较低和资源较富裕的条件下运行多个服务。数据库可靠性工程师首先应该把数据存储分离到独享的服务器中。数据存储所需的工作负载通常对 CPU、RAM 和存储 I/O 有较高要求。有些应用程序是 CPU 密集型或者 I/O 密集型的，但你不希望它们与其他应用程序竞争资源。调优这些工作负载也是非常具体的，因此需要隔离才能正确完成。

当在独享的物理主机上运行数据库时，数据库会与大量组件交互，并消耗这些组件的资源。稍后会简单介绍这些组件。方便起见，这里的讨论基于 Linux 或 UNIX 系统。尽管很

多内容也适用于 Windows，但是系统之间的区别较大，所以不以 Windows 举例。

本章会介绍很多最佳实践，以展示数据库可靠性工程师如何运用所掌握的操作系统和硬件的知识，来解决数据库可用性和性能方面的问题。当谈到抽象的虚拟机和容器时，会适时讲解相关知识。

5.1.2 系统或内核的运维

数据库可靠性工程师应该直接和软件可靠性工程师合作，以定义适合数据库主机的内核配置，这应该成为与数据库二进制文件和相关配置一起自动部署的黄金标准。大多数 DBMS（database management system，数据库管理系统）有厂商特定的要求和推荐配置，故应对其进行评审，确认可用后再应用于生产环境。针对不同的数据库类型，采用的方法也有较大差异，所以下面将讨论共同的关注点。

1. 用户资源的限制

相比普通服务器，数据库使用的资源数量要多得多，包括文件描述符、信号量和用户进程。

2. I/O 调度器

操作系统通过 I/O 调度决定将块级别 I/O 操作提交给存储卷的顺序。默认情况下，这些调度器通常假设使用的是拥有较高寻道延时的旋转磁盘，所以默认使用**电梯调度算法**（elevator algorithm）。该算法尝试根据访问的位置对请求排序，以尽量缩短寻道时间。在 Linux 中，可能有以下选项：

```
wtf@host:~$ cat /sys/block/sda/queue/scheduler
[noop] anticipatory deadline cfq
```

当目标块设备是带有控制器（可以优化 I/O）的一个固态硬盘阵列时，`noop` 调度器是合适的选择。每个 I/O 请求被同等对待，因为固态硬盘的寻道时间相对稳定。`deadline` 调度器通过引入截止时间来防止饥饿，并优先执行读操作，从而最小化 I/O 延时。`deadline` 调度器在多线程、高并发环境（例如数据库负载）中性能更高。

3. 内存分配和碎片

众所周知，数据库是服务器所运行的最耗内存的应用程序之一。要想高效使用内存，需要理解内存是如何分配和管理的。在编译数据库二进制文件时，可以选用不同的内存分配库，示例如下。

- MySQL 5.5 的 InnoDB 引擎采用了定制库，该库封装了 `glibc` 的 `malloc`。GitHub 宣称切换到 `tcmalloc` 之后延时减少了 30%，Facebook 则使用 `jemalloc`。
- PostgreSQL 也使用基于 `malloc` 定制的内存分配库。和其他很多数据库不同，PostgreSQL 以大块的方式分配内存，这种方式称为**内存上下文**。
- Apache Cassandra 2.1 的堆外分配使用 `jemalloc`。

- MongoDB 3.2 默认使用 `malloc` 分配内存，但是可以配置成 `tcmalloc` 或者 `jemalloc`。
- Redis 2.4 使用了 `jemalloc`。

对于大多数数据库负载而言，`jemalloc` 和 `tcmalloc` 被证明可以显著提高并发能力。其性能优于 `glibc` 的原生 `malloc`，同时内存碎片更少。

内存按页分配，默认一页大小为 4 KB。这样，1 GB 内存相当于 262 144 页。CPU 利用**页表**转换地址，每个**页表项**指向一页。TLB（translation lookaside buffer，转换旁视缓冲器，简称快表）是内存缓存，存储了虚拟内存到物理内存最近的转换关系，以便加快检索速度。**TLB 缺失**是指虚拟页到物理页的转换关系不在 TLB 中。对于内存访问而言，在 TLB 中未命中的情况下访问性能会变差，因为地址转换过程会查询页表，这需要多次访问内存。在内存很大、页面很多的情况下，TLB 缺失会导致抖动。

THP（Transparent Huge Pages，透明大页）是一种 Linux 内存管理系统，它通过使用更大的内存页来减少大内存机器上的 TLB 查找开销，从而减少所需的页表项数量。THP 是大小为 2 MB~1 GB 的一块内存。用于 2 MB 页面的表适用于 GB 级别的内存，而用于 1 GB 页面的表最适用于 TB 级别的内存。然而，对这些大页面进行碎片整理会导致严重的 CPU 抖动，这种情况在 Hadoop、Cassandra、Oracle 和 MySQL 等工作负载上都可以看到。要想缓解这种状况，可能需要禁用碎片整理，并且会因此失去多达 10% 的内存。

Linux 并没有针对需要低延迟和高并发性的数据库负载进行专门优化。当内核进入回收模式时，它的行为是不可预测的。我们可以给出的最佳建议之一是，预留一些物理内存，以避免出现停顿和高延时。可以通过配置预留这些内存，避免其被分配。

4. 内存置换

在 Linux 和 UNIX 系统中，**内存置换**是为了减轻内存资源的压力，而将内存中可淘汰的数据存储到磁盘上的过程。与访问内存相比，访问磁盘会慢几个数量级，因此应该在不得已的情况下才这样做。

普遍认为数据库应该避免内存置换，因为它会立即导致延时超出可接受的范围。话虽如此，如果禁用了内存置换，操作系统中的 OOM Killer（Out of Memory Killer）将会结束数据库进程。

对此有两种观点。第一种观点（也是传统的一种）认为，应尽量保证数据库运行，即使有些慢，也好过完全不可用；第二种观点更贴近数据库可靠性工程的观念：延时影响和性能影响同样糟糕，因此内存与磁盘的置换是不可容忍的。

数据库通常有实际的和理论上的内存利用率高水位的相关配置。像缓冲池和缓存这样的固定内存结构，其内存占用大小是固定的，这使得它们可预测。然而，连接层的情况会更复杂一些。有一个基于最大连接数和每个连接内存结构（如排序缓冲区和线程栈）的最大大小的理论上限。使用连接池并做一些合理的假设，有助于预估合理的安全阈值，利用该阈

值可以避免内存置换。

该过程可能会导致配置错误、过程失控等问题，进而耗尽所有内存并最终导致服务器关闭。不过，这是一件好事。如果采用了有效的监控、足够的容量以及故障转移策略，实际上已经避免了因延时而导致违背 SLO 的可能。

> **禁用内存置换**
>
> 仅当有绝对可靠的故障转移流程时，才可以开启内存置换，否则一定会影响应用程序的可用性。

选择开启内存置换，可以降低操作系统将数据库内存置换为文件缓存的概率，但这通常没有帮助。也可以调整数据库进程的 OOM 权重，来降低因其他地方需要内存而导致数据库进程被内核内存分析器结束的概率。

5. 非一致性内存访问

多处理器的早期实现使用称为 SMP（symmetric multiprocessing，对称多处理）的架构，通过 CPU 和内存区之间的共享总线为每个 CPU 提供对内存的平等访问。现代多处理器系统使用 NUMA（non-uniform memory access，非一致性内存访问），为每个 CPU 提供一个本地内存区。内存区和其他处理器之间的访问仍然通过共享总线来完成。如此一来，与访问其他内存区（远端）相比，访问某些内存区（本地）的延时要低很多。

在 Linux 系统中，处理器（包括它的所有核）被视为一个**节点**。操作系统把内存区与本地节点绑定，并基于距离计算节点间的开销。一个进程以及它的线程，会因这种内存访问模式而被分配到一个**首选节点**执行。调度器会临时改变这条规则，但是亲和力（affinity）会始终使用这个首选节点。此外，分配内存之后，进程不会移动到其他节点上。

在大内存结构的环境中（例如数据库缓冲池），这意味着大量内存会分配给首选节点。这种不均衡可能会导致首选节点内存耗尽，没有内存可用。因此，即使与服务器的物理可用内存相比只使用了少量内存，仍然会发生内存置换现象。

Twitter 解决使用 MySQL 过程中遇到的 NUMA 问题

关于这个问题以及 Twitter 的解决方法，Jeremy Cole 发表过两篇重要的文章[1]。最初的方法是通过 `numactl --interleave=all` 强制交叉分配。

通过启用交叉分配，内存实际上分配给所有节点。然而，这也不是完全有效的，因为线上 MySQL 进程在运行一段时间并重启后，操作系统缓冲区缓存可能会满。下面两种方法证明了该过程是可重复和可靠的。

注 1：“A brief update on NUMA and MySQL”和“The MySQL swap insanity problem and the effects of the NUMA architecture”。

- 在 `mysqld` 启动前，通过 `sysctl -q -w vm.drop_caches=3` 刷新 Linux 缓冲区缓存，以使内存分配更公平。即使在守护进程重新启动时，操作系统缓冲区缓存中还有大量数据的情况下也是如此。
- 使用 `MAP_POPULATE`（Linux 2.6.23 以上版本支持），强制操作系统在 MySQL 启动后立即分配 InnoDB 缓冲池，而不是等到内存初始化时。这会强制 NUMA 节点立即做出分配决策，并且这时的缓冲区缓存是干净的（通过前面提到的刷新过程）。

这是数据库可靠性工程师能为软件工程师和软件可靠性工程师提供方便的一个例子。在本例中，对于过度交换导致的问题，需要深入研究操作系统内存管理，并结合对 MySQL 内存管理的深入理解，这样可以更快地修复问题，并且修复合并到 MySQL 的主分支中。

就这一点，对于数据库可靠性工程师的大多数需求而言，可以将内核中的 NUMA 设置为交叉访问模式。相同的问题在使用 PostgreSQL、Redis、Cassandra、MongoDB、MySQL 和 Elasticsearch 时也很常见。

6. 网络

本书假设所有数据存储都是分布式的。网络流量对于数据库的性能和可用性非常重要。网络流量分类如下：

- 节点间流量；
- 应用程序流量；
- 管理流量；
- 备份和恢复流量。

节点间流量包括数据复制、共识和 gossip 协议，以及集群管理。这些数据用于让集群知晓自己的状态，以及保持适当的副本数量。应用程序流量来自应用程序服务器或者代理，用于维护应用状态，支持应用程序新增、修改和删除数据的操作。

管理流量是指管理系统、运维人员和集群之间的通信，包括启动和停止服务、部署二进制程序、数据库和配置变更。当其他地方出现问题时，这是系统的生命线，让我们得以手动或自动恢复系统。顾名思义，备份和恢复流量就是在归档和数据复制、在系统之间移动数据或从备份中恢复数据时所产生的流量。

流量隔离是数据库正确使用网络的第一步，可以通过物理 NIC（network interface card，网络接口卡）或 NIC 分区来实现。现代服务器通常有 1 Gbit/s 和 10 Gbit/s 两种 NIC，可以将它们绑定在一起，以实现冗余和负载均衡。虽然这种冗余将增加 MTBF，但这里是在增强稳健性而非弹性。

数据库需要高效的传输层来管理工作负载。频繁而快速地创建连接、较短的往返路径，以及对延时敏感的查询都需要特定的调优工作，这可以分为以下 3 个方面。

- 增加 TCP/IP 的端口数量，优化大量连接。
- 缩短回收套接字所需时间，以避免大量连接处于 `TIME_WAIT` 状态，从而导致不可用。
- 维护较大的 TCP backlog，避免由于饱和而导致拒绝连接。

TCP/IP 是解决延迟和可用性问题的工具，强烈建议你对其进行深入研究。Douglas E. Comer 所著的《用 TCP/IP 进行网际互连》第一卷（于 2014 年更新）是学习 TCP/IP 的好书和参考材料。

7. 存储

数据库存储是一个庞大的话题，需要考虑单个磁盘、磁盘的分组配置、磁盘控制器、卷管理软件以及其上的文件系统。每一项都有很多内容可讨论，这里只做通览。

图 5-1 展示了访问存储中数据的方式。当从文件读取数据时，需要从用户缓冲区到页面缓存，然后到磁盘控制器，再到磁碟（disk platter）读取数据，最后原路返回将数据交给用户。

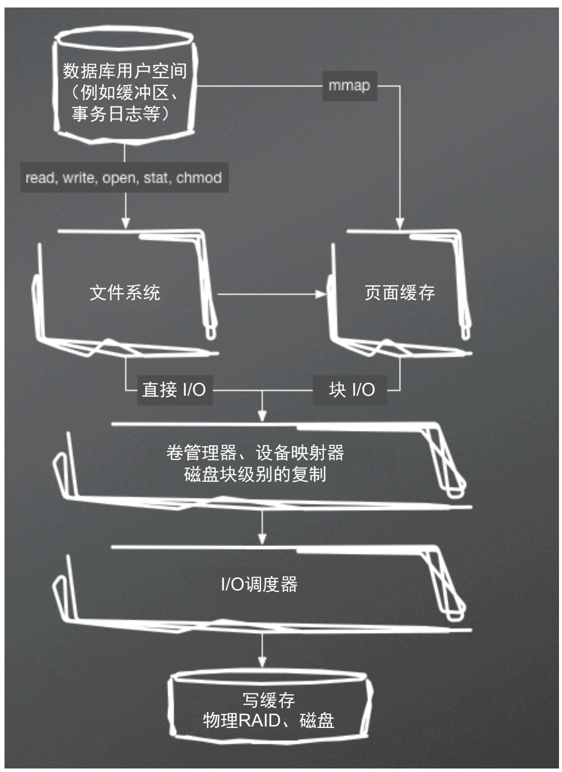


图 5-1：Linux 存储栈

这种由缓冲区、调度程序、队列和缓存组成的复杂级联系统，用于缓解磁盘与内存相比较慢（100 ns 与 10 ms 之别）的状况。

对于数据库存储，主要有 5 个需求（目标）：

- 容量；
- 吞吐量（或每秒 I/O 操作的次数，即 IOPS）；
- 延时；
- 可用性；
- 持久性。

8. 存储容量

容量是数据库中数据和日志可用的存储空间。它们可以存储在大型磁盘、多磁盘条带（RAID 0），或者多个磁盘作为独立挂载点（称为 JBOD 或磁盘簇）的设备上。每种解决方案的故障模式不同。除非采用镜像（RAID 1），否则独立大磁盘存在单点故障。RAID 0 的 MTBF 将降至 1/*N*，其中 *N* 是磁盘数量。JBOD 的故障将更频繁，但与 RAID 0 不同，其他 *N*–1 个磁盘仍然是可用的。一些数据库可以利用这一点，并保持正常运行（以降级模式），直到用新磁盘替换。

不过，数据库对容量的需求只是对存储需求的一部分。如果需要 10 TB 的存储，可以创建一个 10 TB 的条带集，或者在 JBOD 中挂载 10 个 1 TB 的磁盘，并在它们之间分布数据。这也意味着现在有一个 10 TB 的数据库需要立即备份，如果发生故障，需要恢复 10 TB 的数据，这会非常耗时。其间整个系统的容量减少，可用性降低。此外，必须考虑数据库软件、操作系统和硬件能否管理整个数据存储读写的并发负载，以便对单一数据存储进行读写。将该系统分解为更小的数据库，将提高应用程序、备份 / 恢复和复制数据集的弹性、容量和性能。

9. 存储吞吐量

IOPS 是对存储设备每秒 I/O 操作的标准度量，包括读和写。在考虑需求时，必须考虑数据库工作负载峰值的 IOPS，而不是平均值。在规划新系统时，需要估计每个事务所需的 I/O 数量和事务量峰值，这因应用程序而异。对于执行单行插入和读取的应用程序，每个事务中可能执行 4~5 个 I/O，复杂的查询事务可达 20~30 个 I/O。

数据库工作负载往往是混合读 / 写的，并且是随机的，而不是顺序的。当然也有例外，比如只追加写模式（比如 Cassandra 的 SSTable）便是顺序写入。对于机械硬盘，随机 IOPS 主要取决于存储设备的随机寻道时间；对于固态硬盘，随机 IOPS 反而受制于内部控制器和内存接口的速度，这就解释了为何固态硬盘在 IOPS 方面有显著改善。顺序 IOPS 表示磁盘能够产生的最大持续带宽。顺序 IOPS 通常以每秒兆字节（Mbit/s）来表示，指示批量加载或顺序写操作的性能。

在选择固态硬盘时，还需要考虑总线。考虑 PCIe 总线上的闪存产品解决方案，例如具有微秒延迟和 6 Gbit/s 吞吐量的 FusionIO。然而，在撰写本文时，10 TB 将花费约 4.5 万美元。

传统上，与存储容量相比，IOPS 是更重要的制约因素。对于写操作尤其如此，你无法像对待读请求一样通过缓存来优化。通过条带化（RAID 0）或在 JBOD 中添加磁盘的方式，可以像增加存储容量一样，增加存储的 IOPS。RAID 0 可以提供统一的延迟，并消除 JBOD 中可能出现的热点问题，代价是随着条带集中磁盘数量的增加，MTBF 会减少。

10. 存储延时

延时是从客户端发出的端到端 I/O 操作延时，即从发送 I/O 请求到确认读写完成的用时。与大多数资源一样，有请求等待队列，在饱和期间请求可能积压。在某种程度上，排队不是坏事。事实上，许多控制器的设计优化了队列深度。如果工作负载没有足够的 I/O 请求来充分利用可用性能，那么存储可能无法达到预期的吞吐量。

事务性数据库应用程序对 I/O 延时的增加很敏感，因此非常适合选用固态硬盘。通过保持较小的队列长度和较高的卷可用 IOPS 数量，可以保持较高的 IOPS，同时保持较低的延时。持续发送过多的 I/O 请求到一个卷，使其超出可用范围，可能会增加 I/O 延时。

像大型 MapReduce 查询这样的吞吐密集型应用程序，对 I/O 延时的增加不太敏感，因此非常适合用机械硬盘卷。在执行大量顺序 I/O 时，可以通过保持较长的队列来保障机械硬盘卷的高吞吐量。

Linux 页面缓存是延迟的另一个瓶颈。使用 Direct I/O（`O_DIRECT`），可以绕过页面缓存，从而避免几毫秒的延时。

11. 存储可用性

性能和容量是关键因素，但必须考虑存储的可用性。2007 年，谷歌对机械硬盘的故障率进行了一项名为“Failure Trends in a Large Disk Drive Population”的研究（参见图 5-2）。结果显示，前 3 个月预计 100 个磁盘中大约有 3 个发生故障。在前 6 个月正常运行的磁盘中，在接下来的 6 个月中，每 50 个磁盘中大约有 1 个磁盘发生故障。似乎问题不大，但是如果有 6 台数据库服务器，每台服务器有 8 个磁盘，那么在这段时间内可能会有 1 个磁盘发生故障。

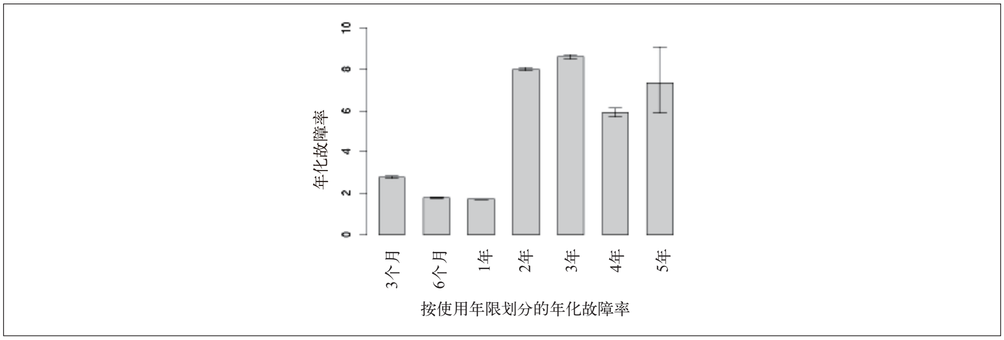


图 5-2：磁盘故障率（来自谷歌的调查）

这就是工程师通过 RAID 1 镜像、数据校验条带集的方式来增强可靠性的原因。本书不讨论奇偶校验的变体（例如 RAID 5），因为写操作的代价非常大。现代存储子系统通常会采用 JBOD，或者按条带（RAID 0）、镜像（RAID 1）或条带镜像（RAID 10）的方式来组织磁盘。鉴于此，很容易推断出 RAID 1 和 RAID 10 对单磁盘故障的容忍度更高，而 RAID 0 更容易导致服务不可用（磁盘条带中的磁盘越多，就越可能发生故障）。JBOD 容忍故障的能力很强，可通过剩余的部分存储来提供服务。

可用性不仅涉及卷的 MTBF，也涉及发生故障后重建所需的时间或 MTTR。选择 RAID 10 还是 RAID 0，取决于在故障期间部署替换数据库主机的能力。出于性能方面的原因，很容易考虑使用 RAID 0。RAID 1 或 RAID 10 的写操作 IOPS 提升 1 倍，如果使用高端驱动器，那么硬件副本可能会变得昂贵。RAID 0 性能高且可预测，但比较脆弱。毕竟，由 5 个机械硬盘组成的条带集，第 1 年的故障率预计约为 10%。如果有 4 个主机，预计一两年就会发生一次故障。

使用镜像，无须重新构建数据库便能替换磁盘。如果正在运行一个 2 TB 的数据库（使用旋转磁盘），那么复制数据库备份所需的时间以小时计，并且可能需要更多时间来同步上次备份的增量。为了适应这种情况，需要考虑在恢复过程中是否有足够的容量来支持峰值流量（在一个甚至两个节点故障的情况下）。在一个数据集只存有部分用户数据的分片环境中，这是完全可以接受的，可以为了节省成本和避免写延迟而采用脆弱的数据卷。

12. 持久性

最后，还要考虑持久性。当数据库在保证持久性的情况下将数据提交到物理磁盘时，它会发出一个称为 `fsync()` 的操作系统调用，而不是依赖页面缓存刷新机制。例如在生成重做日志或预写日志时，必须保证真正写入磁盘，以确保数据库的可恢复性。为了保证写性能，许多磁盘和控制器使用内存缓存。`fsync` 指示存储子系统必须将缓存数据刷新到磁盘。如果写缓存有独立电源，能够确保在断电时数据不会丢失，那么不刷新缓存能提高性能。验证特定的设置确实能够将数据刷新到稳定的存储（无论是 NVRAM、写缓存还是磁盘）非常重要。可以通过 Brad Fitzpatrick 的 diskchecker.pl 工具来验证。

文件系统的操作还可能在发生故障（例如崩溃）时，导致数据损坏和不一致。但是，像 XFS 和 EXT4 这样的日志文件系统，可以显著降低出现这类结果的可能性。

5.1.3　存储区域网络

与直接存储相反，可以使用带有外部接口（通常是光纤通道）的存储区域网络（storage area network，SAN）。SAN 比直接相连的存储设备昂贵得多，通过集中式存储可以减少管理开销，而且灵活性较大。

使用顶级 SAN 可以获得更多缓存。此外，对于大型数据集，还有许多有用的特性，比如

对备份和数据副本进行快照。实际上，数据快照和数据迁移是现代基础设施中的优良特性，其中固态硬盘的 I/O 性能优于传统 SAN。

5.1.4　物理服务器的优点

物理服务器是托管数据库的最简单方法，没有隐藏实现和运行时细节或增加额外复杂性的抽象。在大多数情况下，你可以对操作系统进行尽可能多的控制，并获得尽可能高的可见性，这使得运维相当简单。

5.1.5　物理服务器的缺点

物理服务器也有一些缺点。首先，你可能会浪费专门用于特定服务器的容量；其次，部署这些系统可能相当耗时，而且很难确保每台服务器在硬件和软件方面都是相同的。这就引出了虚拟化。

5.2　虚拟化

在虚拟化系统中，软件分割底层硬件资源，创建各种独享资源。该软件可以让多个应用程序在各自的操作系统上运行，而这些操作系统在同一台物理服务器上运行。例如，在虚拟机出现之后，可以在一台服务器上交替运行 4 个 Linux 实例，每个实例都有专用的计算、内存、网络和存储资源。

利用虚拟化技术，可以将基础设施的资源（包括计算、存储和网络）组合起来，创建可以分配给虚拟服务器的池。这通常称为云计算。在公有云（比如 AWS）上运行服务便是如此。在自己的数据中心也可以这样做。

问题的关键是，无论是采用公有云、私有云还是混合云的解决方案，都可以通过代码定义所需的服务器资源以及操作系统，这样就可以持续部署数据库系统了。这意味着用户可以使用数据库可靠性工程师创建好的标准配置集，部署自己的数据库集群。标准配置主要包括：

- 操作系统；
- 数据库软件的版本；
- 操作系统和数据库配置；
- 安全和权限；
- 软件包和库；
- 管理脚本。

这是一件好事，但在物理资源之上增加一层抽象，管理的复杂性也增加了。下面看看这些问题。

5.2.1 虚拟机管理程序

虚拟机管理程序（hypervisor）或虚拟机监控器的形态，可以是软件、固件或硬件。虚拟机管理程序创建并运行虚拟机。如果一台计算机的系统上运行着一个或多个虚拟机，则这台计算机称为**宿主机**（host machine），其上运行的虚拟机称为**客户机**（guest machine）。虚拟机管理程序为虚拟机操作系统提供虚拟操作平台，并管理虚拟机操作系统的运行。

5.2.2 并发

同一数据库软件在虚拟机上运行时的并发量要比在物理机上运行时小，所以在设计此类虚拟化环境时应主要考虑水平扩展，尽量减少单节点的并发量。

5.2.3 存储

在虚拟化环境中，存储的稳定性和性能会低于预期。在虚拟机的页高速缓冲存储器（Page Cache）和存储控制器之间，存在虚拟的控制器、虚拟机管理程序和宿主机的页高速缓冲存储器，所以 I/O 的延时会增加。出于性能上的考虑，虚拟机管理程序不支持在写操作时调用 `fsync`，这意味着在应用程序崩溃时不能保证数据真正落盘。

此外，即使可以在 10 分钟甚至更短的时间内新建一个虚拟机，也不一定能创建出数据库正常工作所需的数据。比如部署了一个新副本，需要从别处同步原始数据。

虚拟化环境中的存储主要分为两类：本地存储和持久化块存储。本地存储的数据是易失的，这类数据只有在虚拟机生命期内才可访问。持久化块存储设备可以挂载到任何虚拟机上，挂载后所有虚拟机都可以对这些数据进行读写。即使该关闭虚拟机，该块存储设备还可以挂载到其他虚拟机上。对于数据库而言，这类外部的持久性存储是理想的数据存储介质。通常这类块存储设备也支持快照功能，以便数据迁移。

这类块存储设备比传统的磁盘更依赖网络，因此网络拥塞会大大降低存储性能。

5.2.4 用例

有了上述注意事项，在计划将虚拟化和云资源用于数据库基础设施时，数据库可靠性工程师需要更谨慎地思考。在设计这类基础设施时，必须将上述因素都考虑在内，总结如下。

- 不能保证持久性意味着数据丢失不可避免。
- 实例不稳定意味着自动化、故障转移和故障恢复必须非常可靠。
- 水平扩展要求可以自动化地管理大量服务器。
- 应用必须能容忍延时不稳定的问题。

尽管有许多不足，但建立基于虚拟化和云的数据库基础设施仍然非常有价值。而创建用户

可以在其上进行构建和使用的自助服务平台，可以扩大数据库可靠性工程师的影响。即使数据库可靠性工程师人数不多，也可以传播知识和最佳实践。

快速部署有助于对应用程序和原型开展详细的测试，也有助于提高开发团队的效率，因为无须依赖数据库可靠性工程师来完成部署和配置工作。这意味着，在部署新应用程序的持久化层时，开发人员可以自行部署，无须数据库可靠性工程师介入。

5.3 容器

容器位于物理服务器及其操作系统之上，所有容器共享物理机的操作系统内核、二进制文件和库。这些共享的组件是只读的，每个容器只操作自己单独的挂载点。容器比虚拟机更轻量，实际上，它们只有兆字节大小。新建虚拟机需要 10 分钟左右，而容器只需几秒就能启动。

然而，对数据存储而言，Docker 快速启动的优点常常因需要挂载、引导和同步数据被抵消了。此外，定制化内核、过重的 I/O 负载以及网络拥塞，使得共享宿主机 / 操作系统的模型面临很大的挑战。Docker 快速部署的优点使其成为开发和测试环境中的强大工具，也是数据库可靠性工程师的实用工具。

5.4 DaaS

越来越多的公司在为虚拟化和云服务寻找第三方方案。拿第 6 章将讨论的自助服务模型来讲，最终采用的是第三方管理的数据库平台。所有公有云都提供这类服务，其中最著名的是亚马逊 RDS（relational database service，关系型数据库服务）。RDS 涵盖 MySQL、PostgreSQL，Aurora、SQL Server 以及 Oracle。在这类环境中，可以为基础设施选择已经部署好的数据库系统。

由于很多简单的操作已经自动化，节省了许多宝贵的工程资源，因此当前 DaaS 被广泛采用。DaaS 提供的典型功能包括：

- 部署；
- 主库故障迁移；
- 补丁和升级；
- 备份和恢复；
- 度量值展示；
- “特殊机制”带来的高性能，比如亚马逊的 Aurora。

上述功能能够节省时间，但软件工程师切勿以为不再需要数据库专家了，事实远非如此。抽离的服务自有其挑战，更重要的是，它们便于你发挥专长，创造最大价值。

5.4.1 DaaS面临的挑战

缺少可视化工具是 DaaS 面临的最大挑战之一。由于无法访问操作系统、网络设备和硬件，很多重要的问题将无法诊断。

> **DaaS 与网络时间协议**
>
> 我们曾使用一个知名的 DaaS 服务，而我们的客户决定使用厂商提供的新版本的数据库。我们劝说客户不要使用，但厂商给予了一些优惠并且我们的服务仍处于测试阶段。在这种情况下，运维人员忽略了在所有节点上执行网络时间协议（network time protocol，NTP）来同步时间。经过数小时的排查，仍然无法解释副本滞后的原因，于是我们给厂商的技术支持打电话试图弄清楚状况。真是糟糕的一晚。

尽管现在许多监控系统在 TCP 级别上收集数据库 SQL 数据，以便管理大规模的数据收集，但又不得不使用日志或内部快照，比如 MySQL 中数据的 performance schema。另外，很多经典的追踪和监控工具无法使用，比如 `top`、`dtrace` 和 `vmstat` 等。

DaaS 和其他虚拟化环境一样存在持久化问题，并且重要组件（比如复制和备份）的实现通常是**黑盒**，取决于厂商。

5.4.2 数据库可靠性工程师与DaaS

在市场上，DaaS 的卖点是让你无须聘请费用高昂且难以雇用 / 留住的数据库专家。DaaS 平台确实能让你更快地引入从运维角度来说可靠的数据库基础设施，进而推迟雇用或聘请数据库专家，但还远达不到不需要聘请数据库专家的程度。

如果有什么不同的话，那就是 DaaS 将复杂而容易解决的问题抽象了出来，因而在数据库专家加入之前，你可能会碰到难以解决的问题。除此之外，还有许多事情需要尽早决策，而做出这些决策需要深入理解所选择的数据库引擎。这些重要决策包括：

- 使用哪个数据库引擎；
- 如何对数据建模；
- 合适的数据访问框架；
- 数据库安全性决策；
- 数据管理、容量及增长规划。

所以，即使 DaaS 可以帮助软件工程师完成更多工作，数据库可靠性工程师也需要努力帮助他们做出正确的选择，利用自身专业知识帮助成功部署 DaaS。

DaaS 对组织很有吸引力，尤其是在组织发展的早期，那时候时间非常宝贵。在这样的环境中，强烈建议数据库可靠性工程师为基础设施准备好数据迁移和灾难恢复方案。可以适

时将 DaaS 解决方案所做的事情自动化，以便完全控制数据存储，并且可以随时掌握数据存储状况。

5.5 小结

本章介绍了主机类型选项——物理机、虚拟机、容器以及服务，讨论了处理器、内存、网络和存储资源所产生的影响，以及资源不足和配置错误所产生的影响。

第 6 章将讨论如何通过适当的工具和过程来管理这些数据库基础设施，以实现规模化，并管理风险和故障。内容将涵盖配置管理、编排、自动化和服务发现以及管理。

第 6 章

基础设施管理

第 5 章研究了可以运行数据存储的各种基础设施组件和范例。本章讨论如何管理具有一定规模的这些环境。首先讨论最小的单元——单个主机的配置和规格；然后讨论主机的部署和组件之间的编排；之后讨论动态发现基础设施的状态，以及状态信息的发布——服务发现；最后讨论开发所用到的技术栈，以及如何创建大型生产系统对应的开发环境。

轻松地手动管理一两台服务器，而且服务稳定运行的时代已成历史。现在必须做好准备，以较少的人手来维护大规模、复杂的基础设施。自动化对于确保重复且可靠地部署数据存储至关重要，关乎应用程序的稳定性和可用性，以及新功能的部署速度。我们的目标是消除重复和手动的流程，并通过标准化流程以及自动化，创建可轻松重建的基础设施。

哪些适合自动化呢？

- 软件安装，包括操作系统、数据库以及相关软件和工具包。
- 根据软件功能和负载需求，配置软件和操作系统。
- 为新建的数据库初始化数据。
- 安装相关工具，比如监控客户端、备份软件和运维工具。
- 测试基础设施的配置和功能。
- 静态的和动态的合规性测试。

归根结底，我们的目标是可以持续地构建和重建数据库基础设施中的任何组件，以确保在排查问题和测试时，可获知各组件当前和过去的状态。

诚然，这只是概述，若想深入了解相关内容，建议参考基夫·莫里斯所著的《基础设施即

代码》[1]。本章主要讲解如何通过代码管理基础设施的各种组件，以及如何简化数据库可靠性工程师的工作。

6.1 版本控制

为了实现上述目标，要在此过程中控制所有组件的版本，包括以下几项：

- 源码和脚本；
- 依赖的软件库和包；
- 配置文件和相关信息；
- 操作系统镜像和数据库二进制文件的版本。

VCS（version control system，版本控制系统）是所有软件工程流程的关键。数据库工程师、系统工程师和软件工程师一起使用 VCS 来构建、部署和管理应用程序和基础设施。然而传统上系统工程师和数据库工程师往往不使用任何 VCS，即使碰巧使用了某个 VCS，也可能与软件工程师使用的不同，导致基础设施的版本和代码的版本不对应。

流行的 VCS 包括：

- GitHub；
- BitBucket；
- Git；
- 微软的 Team Foundation Server；
- Subversion。

VCS 必须是基础设施中所有内容的真实来源，包括脚本、配置文件以及用于创建数据库集群的定义文件。当需要新增内容时，必须将其添加到 VCS 中。在修改内容时，需要先签出然后进行修改，再将更改签入 VCS 中。签入完成后，才能进行评审、测试，最后就可以部署了。值得注意的是，在将密码存储在 VCS 中之前，应先对其进行脱敏。

6.2 配置定义

为了定义配置和构建数据库集群的方法，需要使用一系列组件。脚本可能使用的是 Windows PowerShell、Python、Shell Scripts、Perl 或者其他语言，但配置管理软件可能使用 DSL（domain-specific language，领域特定语言）。流行的配置管理软件有：

- Chef；
- Puppet；
- Ansible；

注 1：本书中文版已由人民邮电出版社出版，详见 https://www.ituring.cn/book/1879。—— 编者注

- SaltStack；
- CFEngine。

通过定义配置而不是编写脚本，可以创建易读的组件，并在整个基础设施中复用。定义配置可以保证一致性，通常还能减少向配置管理中添加新组件的工作量。

这些应用程序具有原语（primitive），在 Chef 中叫 recipe，在 Puppet 中叫 manifest，它们汇集成 cookbook 或 playbook。为了满足不同的需求，比如测试环境与生产环境、文件分发方案，以及依赖库或者模板的扩展，这些 cookbook 或 playbook 将 recipe 汇总，而 recipe 记录了用于覆盖默认值的属性。playbook 的最终形态是生成基础设施特定组件的代码，比如安装 MySQL 或者 NTP。

定义文件本身可能会变得非常复杂，所以应该共用某些属性。需要把这些属性参数化，以便根据输入在不同的环境（例如开发环境、测试环境和生产环境）中运行相同的定义文件。由这些定义及其应用程序执行的操作，必须是**幂等**的。

> **幂等**
>
> 幂等操作指重复执行该操作，输出结果始终相同。幂等操作有预期状态，不管组件的当前状态如何，幂等操作都会使该组件进入预期状态。比如更新配置文件以设置缓冲区缓存的大小，可以假设该配置存在，但这种假设偏于简单且容易出错。或者，可以向配置文件中插入一行，如果该行已经存在，就会导致重复，该操作就不是幂等的。
>
> 应该先用脚本检查该配置是否存在，如果该配置已经存在，则可以对其进行修改；如果配置不存在，就插入一行，该操作就是幂等的。

可以将一个分布式系统（例如 Cassandra）所需的配置定义分为如下几类。

- 主要属性：
  - 安装方法、位置以及散列算法；
  - 集群名称和版本；
  - 用户组和用户权限设置；
  - 堆大小；
  - JVM 配置与调优；
  - 目录布局；
  - 服务配置；
  - JXM 配置；
  - 虚拟节点。
- JBOD 配置和分布。
- 垃圾收集行为。

- 种子节点发现。
- YAML 文件的配置信息。
- 操作系统资源的配置。
- 外部服务：
  - PRIAM；
  - JAMM（Java 度量指标）；
  - 日志；
  - OpsCenter。
- 数据中心与机架布局。

除了幂等和参数化，所有组件都需要在上线前后进行适当的测试，并将对应的测试集成到监控系统和日志管理系统中，以便进行错误管理和持续改进。

上线前后的测试包括验证启动和停止状态是否符合预期，其他测试着重于变更可能会影响的功能，检查功能是否符合预期。在确定变更能否实施之前，我们假设已经通过了运维测试、性能测试、容量测试以及负载测试，测试主要验证功能是否符合预期。关于幂等的真实案例，可参考 Salesforce 开发者博客 [2]。

6.3 基于配置的构建

定义了服务器规格、验收测试以及用于自动化和构建的模块之后，便拥有了构建数据库所需的所有源代码。有两种方式可以做到这一点：**烘焙**（baking）和**油煎**（frying）。看到“烘焙”“油煎”“菜单”“主厨”等，是否感觉有点饿了？相关信息可以参考 John Willis 的文章“DevOps and Immutable Delivery”。

frying 涉及在主机部署时完成动态配置。创建好主机、操作系统安装完成，就可以进行动态配置了。前面提到的配置管理应用程序可以通过 frying 构建和部署基础设施。

例如通过 frying 创建 MySQL Galera 集群，可能会看到如下内容。

- 服务器硬件已就绪（3 个节点）。
- 操作系统安装完成。
- Chef Client 和命令行工具已安装，cookbook 上传成功。
- cookbook 中修改操作系统权限和相关配置已生效。
- cookbook 中默认依赖包安装已生效。
- 已经新建并上传了包含集群节点属性（比如 IP、节点初始化、包名称）的 DataBag。
- 节点角色（Galera 节点）已生效：
  - MySQL/Galera 二进制包安装完成；

注 2：Implementing Idempotent Operations with Salesforce.

- MySQL 工具包及脚本安装完成；
- 设置基本配置；
- 执行测试；
- 服务启动和停止；
- 集群创建 / 主节点设置；
- 剩余节点设置；
- 执行测试。

• 将集群注册到基础设施中。

baking 的过程包括获取基础镜像，以及在构建时配置该基础镜像。该过程会创建“黄金镜像”，该镜像用作为相同任务构建所有主机的标准模板。可以为该镜像生成快照，保存起来供以后使用。AMI（Amazon Machine Images）或者虚拟机镜像就是 baking 过程的成果。在此过程中，没有什么是动态的。

Packer 是来自 Hashicorp 的镜像创建工具。Packer 可以为不同环境（例如亚马逊 EC2 或 VMware）创建镜像，大多数配置管理工具也能创建 baking 镜像。

6.4 维护配置

在理想情况下，配置管理有助于缓解甚至消除配置漂移。配置漂移是在完成 baking 和 frying，以及部署之后发生的。尽管这些组件的所有实例初始状态完全相同，但是人们不可避免地要登录、调试、安装软件或者进行一些实验，因此会留下一些痕迹。

不可变基础设施是指在部署后不能更改或者修改的基础设施。如果必须要做一些变更，应该通过版本控制的配置定义来完成，然后重新部署服务。不可变基础设施很有吸引力，因为具有如下特性。

❑ **简单性**

因为不允许修改，所以基础设施的可能状态将大大减少。

❑ **可预测性**

状态总是已知的，排查和发现问题会非常快，并且在定位问题时易于重现。

❑ **可恢复性**

通过重新部署黄金镜像，易于恢复到刚部署时的状态，这可以显著缩短 MTTR。这些镜像是已知并经过测试的，可以随时部署。

话虽如此，不可变基础设施的开销可能非常大。例如拥有 20 个节点的 MySQL 集群需要修改一个参数，在签入修改的配置之后，为了让变更生效，需要重新部署集群中的每个节点。

折中起见，可以允许那些频繁、自动化且可预测的变更，而仍然禁止手动变更，这样可以保留可预测性和可恢复性，同时减少运维开销。

配置定义的实施

如何实施这些策略呢？

1. 配置同步

前面提到的许多配置工具提供同步功能，这意味着任何修改都将被覆盖，这些配置会强制覆盖成标准配置。但是，这依赖于状态完整地同步，如果漏掉了某些区域，就会导致配置漂移。

2. 组件重新部署

借助合适的工具能发现配置的差异，并通过重新部署来彻底消除差异。有些环境可能需要经常重新部署，或者在某组件上发生手动登录 / 交互后重新部署。在这种情况下，通常 baking 解决方案更适用，因为该方案消除了部署后进行配置的开销。

配置的定义和管理，有助于确保正确构建每个服务器或者实例，并保持状态不变。部署过程有更高层的抽象——跨服务基础设施定义和部署编排。

6.5 基础设施定义和编排

前面研究了单机配置和部署，无论是物理服务器、虚拟机还是云主机。下面看看主机组，毕竟很少会单独管理一个数据库实例。假设我们一直在使用分布式数据存储，那么需要能够一次构建、部署和操作多个系统。

用于部署基础设施的编排和管理工具与部署（frying 或 baking）应用程序集成，以创建完整的基础设施，这包括可能不使用主机的服务（例如虚拟资源或平台即服务的配置）。这些工具为通过代码来创建整个数据中心或服务提供了理想的解决方案，使得开发人员和运维人员能够从头到尾地构建、集成和启动基础设施。

通过将基础设施抽象为可归档、通过版本进行控制的代码，这些工具可以与配置管理应用程序集成，用于自动化部署主机和应用程序。同时为自动化工具设置好必要的底层基础设施资源和服务，以便其能高效地完成相应的工作。

在讨论基础设施定义时，经常使用术语**栈**。你可能听说过 LAMP 栈（Linux、Apache、MySQL、PHP）或 MEAN 栈（MongoDB、Express.js、Angular.js、Node.js），这些都是通用解决方案栈。特定的栈可能针对特定的应用程序或一组应用程序。在讨论通过工具进行自动化和编排基础设施的定义时，栈甚至有更具体的含义，后面会引用这里的定义。

这些栈的结构会对团队中数据库可靠性工程师履行职责产生重大影响。接下来讨论各种栈结构及其对数据库可靠性工程师角色的影响。

6.5.1 单一基础设施定义

在该栈中，组织的所有应用程序和服务都在一个大文件中集中定义。换言之，所有数据库集群都将在同一文件中定义。在多个应用程序和服务使用一个或多个数据库的典型环境

中，可能有多个应用程序及其关联数据库在同一文件中定义。

单一基础设施定义确实没有优点，倒有很多缺点。从编排 / 基础设施即代码的全局角度来看，可以将问题阐释如下。

- 如果要更改定义，需要测试整个定义。这意味着测试是缓慢而脆弱的，将促使人们避免变更，从而导致基础设施僵化且脆弱。
- 变更也更容易破坏所有内容，而不是限制在基础设施的一个组件之中。
- 构建测试环境或开发环境时，不得不将所有内容构建在一起，而不是单独构建部分组件，后者更易于聚焦。
- 变更通常仅限于了解整个栈的少部分人，这会成为变更的瓶颈并且影响迭代速度。

如果团队引入了 Terraform 之类的新工具，并改用这些工具重新部署和配置整个基础设施，就会发现处于整体定义的栈中。在考虑基础设施定义时，会有水平拆分（一个栈中的各层）和垂直拆分的需求（根据功能划分栈，以便一个服务使用一个栈，而不是所有服务共用一个栈）。

6.5.2 垂直拆分

如图 6-1 所示，通过将定义拆分为每个服务独有一个定义文件，可以减少定义的大小，降低复杂性。因此，之前一个定义文件可以拆分为两个。这样可以缩小仅更改一项服务时的故障域，测试工作量随之减半，相应地缩小开发环境和测试环境的规模。

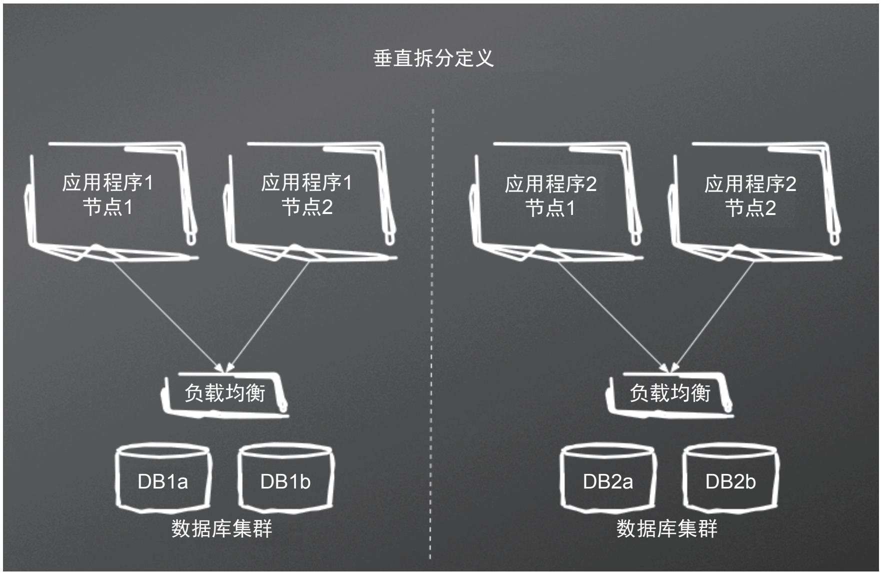


图 6-1：垂直拆分

如果有多个应用程序使用同一数据库层（这很常见），将更具挑战性。此时需要创建 3 个定义，一个是共享数据库定义，另外两个是单独的服务定义，不包括数据库层，如图 6-2 所示。

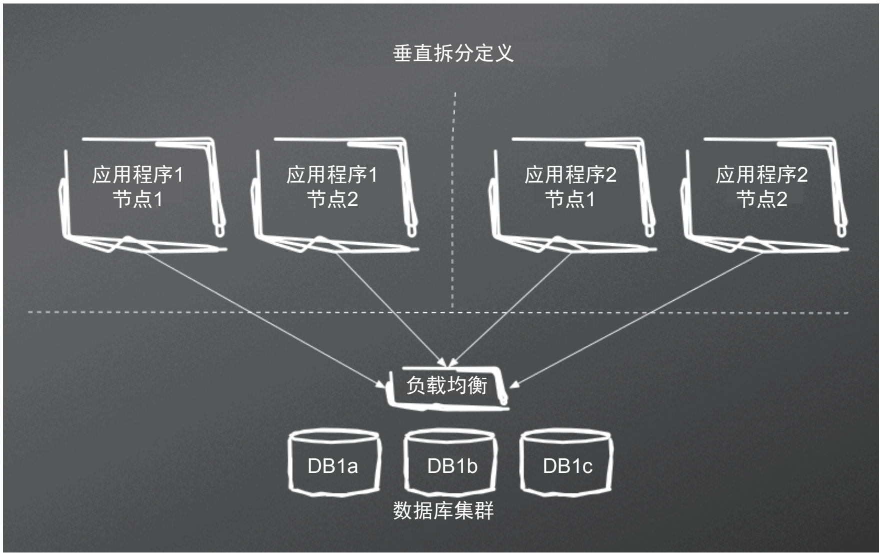


图 6-2：在共享数据库情况下，垂直拆分服务定义

现在，定义文件变更的故障域进一步缩小，因为每个定义更小且更集中。但是，其中仍有两个应用程序与数据库耦合，需要进行集成测试，以确保数据库层的变更和另外两个应用程序栈兼容。因此，测试仍然需要构建和部署所有应用程序。拆分应用程序后，能否串行或并行地构建和测试应用程序，取决于基础设施的约束。

6.5.3 分层（水平定义）

如果采用已拆分的应用程序定义，则可以按层划分定义文件，也称**水平拆分**。因此，标准的 Web 应用程序可能包含 Web 服务器栈、应用程序服务器栈和数据库栈。从该层面拆分基础设施定义的主要优势是，进一步缩小了故障域。换言之，如果需要更改数据库服务器的配置，无须担心可能会破坏 Web 服务器的构建。

在垂直拆分和水平拆分之后，将面临新的、有趣的复杂性。具体而言，栈之间的通信需要共享数据。数据库负载均衡的虚拟 IP 必须与应用程序服务器共享，但服务栈有自己的定义。这种动态基础设施需要一个**服务目录**，以确保基础设施的任何组件都能与其他组件高效共享状态，以便进行通信和集成。

6.6　验收测试和合规性

利用自动化和基础设施即代码益处颇多，但我们还未提及验收测试和合规性。有了基础设施的镜像，就可以利用 ServerSpec 之类的工具（使用描述性语言）来定义基础设施镜像的测试。这有利于对基础设施实践 TDD（test-driven development，测试驱动开发），进而提供了将基础设施和软件工程相结合的绝佳机会。

借助 ServerSpec 之类的自动化测试框架，可以深入研究合规性和安全性，这是 ServerSpec 的用武之地，这样就可以创建针对数据库安全性和合规性的测试套件了。InSpec 是 ServerSpec 的一个插件，可堪此用。

6.7　服务目录

随着动态地自动构建、扩展和销毁环境，所有基础设施组件都能使用的当前状态必须保存在一个数据源中。服务发现是一种抽象，它将服务和负载均衡的特定名称和端口映射到语义名称。例如 mysql-replicas 是一种语义名称，它可以包含任意数量的 MySQL 主机，这些主机从主库复制数据。服务发现的作用是能够通过语义名称而不是 IP 地址或主机名进行引用，服务目录中的信息可以通过 HTTP 或 DNS 访问。

撰写本文时，常见的服务发现工具有：

- Zookeeper；
- Consul.io；
- Etcd；
- 自研工具。

对于这样的服务目录，数据库可靠性工程师可能见过不少案例，下面列出一些（后面探讨具体架构时会详述）。

❑ **数据库故障转移**

　通过向注册中心登记**可写服务器的** IP，可以为负载均衡创建模板。当 IP 发生切换时，将重新生成负载均衡的配置并加载。

❑ **分片**

　向应用程序主机提供可写分片的信息。

❑ **Cassandra 种子节点**

　告知正在启动的节点到哪里寻找种子节点。

服务目录可以非常简单，仅仅记录服务相关的数据以便集成，也可以包含其他许多功能，

例如“健康检查”的数据，这样通过服务目录就可以判断实例是否正常工作，还可以在注册目录存储键值对[3]。

6.8 完成拼图

在较高的层次上有许多信息，下面以在 MySQL 中数据库可靠性工程师日常如何使用这些概念为例说明。简单起见，假设在亚马逊 EC2 环境中运行。你的任务是为用户的主数据库（集群是分片的，并且需要扩容）搭建一个 MySQL 新集群，工具如下：

- MySQL 5.6 社区版；
- 用于复制管理和故障转移的 MySQL MHA；
- 用于保存集群状态的 Consul。

当然，用于用户数据库 MySQL 分片的 Terraform 和 Chef cookbook 文件已经签入 GitHub 了。构建过程应该不需要任何手动更改。你可能会抱怨，应该将容量分析和数据库分片的部署也自动化，但这只是你开展工作的第 3 个月，还没来得及这么做。

检查部署日志，将看到上次部署的 MySQL 分片情况，并将其版本与 Terraform 和 Chef 代码的当前版本做比较，以确认此后没有任何变更。一切就绪后，可以开始部署了。首先，以 plan 选项运行 Terraform，以显示执行路径并验证不会发生任何意外。假设一切顺利，可以继续执行 Terraform 命令来构建分片。

Terraform 执行 chef provisioner，通过查询 Consul 获得最新的 `shard_id`，并增 1 作为新的 `shard_id`，包含以下步骤。

1. 选择合适的 AMI，在两个可用区域中启动 3 个 EC2 实例，作为 MySQL 分片的主机。
2. 在这些主机上配置 MySQL，并启动服务。
3. 在 `shard_id` 命名空间下，将每个节点注册到 Consul。
    - 第一个节点注册为 master，随后注册的节点用作故障转移。
    - 启动数据复制过程。
4. 选择适用于 MySQL MHA 管理器的 AMI，在两个可用区域中启动两个 EC2 实例。
5. 在 `shard_id` 命名空间下，将 MHA manager 注册到 Consul。
6. 利用 Consul 的节点数据配置 MHA。
7. 启动 MHA replication manager。
8. 执行一系列故障转移测试。

这样就在 Consul 中注册了由 MHA 管理的 MySQL 集群，此后有些任务会自动执行，包括：

注 3：Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, et al. Locating Data Sources in Large Distributed Systems.

- 使用 Consul 的备份脚本，开始自动对主服务器进行快照；
- 位于 AMI 中的监控代理，开始自动向监控系统推送度量值和日志。

最后，如果觉得没问题，在 Consul 中把数据库分片标记为 active。代理服务器会将其添加到配置文件并重新加载，应用程序服务器将其标识为可用状态，并开始向其发送数据。

6.9　开发环境

在开发环境或沙箱中，本地测试对于该工作流程来说非常重要。在将变更签入 VCS 之前，应该清楚其影响。可重复性是本章的主要目标之一，这意味着沙箱在软件和配置上必须尽可能接近实际环境，即操作系统、配置管理、服务编排，甚至服务目录都相同。

前面讨论部署时提到了 Packer。Packer 能够以相同的配置创建多种镜像，包括**工作站**虚拟机镜像。在工作站上使用 Vagrant 之类的工具，可以下载最新的镜像、构建虚拟机，甚至运行标准测试套件，以验证所有功能均按预期运行。

完成变更，进行了测试，并对 VCS 中任何新变更再次测试之后，就可以将这些变更直接签入团队的 VCS 中，为集成和部署做准备。

6.10　小结

使用基础设施即代码、自动化和版本控制是可靠性工程师的必备技能，数据库可靠性工程师自然也不例外。运用本章所讲的工具和技术，可以为工程团队消除琐事、减少错误并创建自助服务的部署系统。

第 7 章将深入研究基础设施的关键组成部分：备份和恢复。数据库层的一个显著特点是，数据持久性和可用性至关重要。尽管大多数其他环境可以作为组件进行构建并快速、轻松地部署，但数据库需要安全地维护和管理大量数据。有通用的工具可以用于此操作，这正是接下来要讨论的内容。

第 7 章
备份和恢复

第 5 章和第 6 章重点讨论了基础设施的设计和管理，介绍了如何构建、部署和管理运行数据库的分布式基础设施，包括快速添加节点以扩容或替换故障节点。本章讨论重要的问题：数据备份和恢复。

大多数人认为数据备份和恢复枯燥乏味，是辛劳而无聊的工作。这类工作通常由初级工程师、外包商，以及团队不爱用的第三方工具来承担。我们以前使用过一些糟糕的备份软件，对这一点深有同感。

尽管如此，备份与恢复仍然是运维中最重要的流程之一。在节点之间、数据中心之间移动数据，以及对数据进行归档，是企业最宝贵的资产（数据）的持续活动。应高度重视数据备份与恢复，而不是将其视为运维的二等公民。不仅要了解数据恢复的目标，还应该熟悉操作和监控流程。DevOps 思想倡导每个人都应该有机会编写代码，并将代码部署到生产环境中。建议每个工程师至少参与一次关键数据的恢复流程。

作为满足实际需求（恢复）的一种手段，我们创建并存储数据副本（也称备份和存档）。有时恢复是美好而轻松的，例如配置一个审计环境或备用环境。然而通常情况下，恢复是为了快速替换故障节点，或者向现有集群中添加节点以扩容。

如今，在分布式环境中，备份和恢复领域存在新的挑战。一如往常，大多数本地数据集大小适中，最多几 TB。不同的是，这些本地数据集只是分布式数据集的一部分，恢复一个节点相对容易，但是在集群中保持状态更具挑战性。

7.1　核心概念

首先介绍备份和恢复的核心概念。对于有经验的数据库工程师或系统工程师，其中一些知识可能比较初级。若是如此，可选择跳过相关内容。

7.1.1　物理备份与逻辑备份

物理上，备份数据库是备份数据所在的文件。这意味着保持数据库的特定文件格式。数据库中通常有一组元数据，用于定义包含哪些文件以及数据库结构。如果你备份了文件，并且希望另一个数据库实例能够使用这些文件，则需要备份并存储数据库所依赖的相关元数据，以便备份具有可移植性。

逻辑备份是将数据库中的数据导出为另一种格式，理论上这种格式可以移植到任何系统。通常仍会有一些元数据，但这种备份更关注备份的时间点。例如导出所有插入语句，用这些语句填充空数据库使其变更到最新状态，或者以 JSON 格式保存每一行。因此，逻辑备份往往非常耗时，因为需要逐行提取数据，而不是通过物理副本和写操作复制。类似地，恢复涉及数据库的所有常规开销，比如加锁、重做或撤销日志。

这种二分法的一个例子是，基于行的复制和基于语句的复制之间的区别。在许多关系型数据库中，基于语句的复制意味着提交时会追加一个 DML（data manipulation language，数据操作语言，也称插入、更新、替换、删除）语句的日志。这些语句同步到副本，并在副本中重放。另一种复制方法是基于行或 CDC（change data capture，数据变更捕获）。

7.1.2　脱机备份与联机备份

脱机备份（或冷备份），是指在关闭数据库实例的情况下进行备份。这样在没有其他进程读写数据的情况下，无须担心维护某个时间点的状态，即可快速复制文件。这是一种理想的工作状态，但非常罕见。

在联机备份（或热备份）中，仍然需要复制所有文件，但要先获取一致的时间点数据快照（包含备份开始时已存在的所有数据），这样就增加了复杂性。此外，在备份期间如果有访问数据库的实时流量，还必须注意不要让其超过存储层的 I/O 吞吐量。即使有所限制，用于保持一致性的机制也会大大增加应用程序的延迟。

7.1.3　全量备份、增量备份和差量备份

无论采用哪种方法，全量备份都意味着完整备份整个本地数据集。对于较小的数据集而言，全量备份的耗时微不足道；但对于 10 TB 的数据，可能会非常耗时。

差量备份指仅备份上次全量备份以来变更的数据。在实践中，由于数据采用页之类的结构，所以通常备份的数据量比变更的多。页有特定的大小，比如 16 KB 或 64 KB，其中包

含多行数据。增量备份将备份任何数据已修改的页，因此，较大的页需要备份的数据比变更的数据多得多。

增量备份类似于差量备份，只是会用最后一次备份的数据（增量备份或全量备份）作为查找变更数据的时间点。因此，如果要恢复增量备份，则可能需要恢复最后的全量备份，以及一个或多个增量备份，以到达当前时间点。

了解这些概念之后，下面讨论在选择备份和恢复的有效策略时需要考虑的各种问题。

7.2 恢复的考量

第一次评估一个有效的策略时，应该考虑 SLO（见第 2 章）。具体而言，需要考虑可用性和持久性指标。任何策略都需要你在预定义的正常运行时间内恢复数据。而且，备份需要足够快，以满足数据持久性需求。如果每天备份，并且两次备份之间的事务日志仍然保存在节点级别的存储中，则在下一次备份之前这些事务日志很可能会丢失。

此外，还需要考虑数据集在整个生态系统中的功能。例如订单可能存储在一个关系型数据库中，所有内容都在事务中提交，因此很容易针对数据库中的其他数据进行恢复。但是，在数据库中生成订单之后，存储在消息队列或键值对存储系统中的事件可能会触发工作流。这些系统可能是最终一致的，甚至可能是易失的，依赖关系型数据库进行引用或恢复。在恢复时，如何考虑这些工作流？

在快速开发的环境中，备份中存储的数据可能是由某个版本的应用程序写入和利用的，而恢复之后运行的是另一版本的应用程序。应用程序如何与旧数据交互？若数据有版本信息，那么很好。但是你必须了解一点，并做好应对准备；否则，逻辑上应用程序可能会破坏数据，并造成更大的问题。

在规划数据恢复时，必须考虑其中每一个因素，以及许多无法预料的其他因素。如第 3 章所述，我们不可能对所有可能发生的事情都做好准备。但是，这是一项关键服务，保证数据可恢复是数据库可靠性工程师最重要的职责之一。因此，对数据可恢复性的规划必须尽量全面，并考虑尽可能多的潜在问题。

7.3 恢复场景

鉴于此，下面讨论可能需要恢复的事件和操作类型，以便满足各种需求。首先划分为计划内的场景和计划外的场景。如果把恢复视为应对紧急情况的工具，会限制团队仅在紧急情况和模拟演练中使用该工具；而如果能将恢复纳入日常工作，就可以在紧急情况下更熟练地执行恢复流程，提高成功率。类似地，我们将有更充足的数据来确定恢复策略是否符合 SLO。通过日常多次执行，可以更容易地获得样本集，该样本集可以包含上限，并且可以用某种程度的确定性来表示，以用于规划目的。

7.3.1　计划内的恢复场景

需要考量的日常恢复需求是什么？列举如下：

- 构建新的生产节点和集群；
- 构建不同的环境；
- 数据的提取、转换和加载，以及下游数据存储流水线；
- 运维测试。

在执行这些操作时，请确保将流程添加到运维可见性系统中。

❑ **时间**

运行每个组件以及整个流程需要多长时间？解压？复制？应用日志？测试呢？

❑ **大小**

备份在压缩和未压缩的情况下有多大？

❑ **吞吐量**

硬件承受了多大的压力？

这些数据有助于提前处理容量问题，从而确保恢复流程是可行的。

1. 构建新的生产节点和集群

无论数据库是否是不可变基础设施的一部分，都有可能定期重新构建。当然，这会利用恢复流程。很少将数据库设置为自动伸缩，因为启动一个新节点并将其加入集群比较耗时。团队应该确定一个日程，定期将新节点加入集群，以测试这些流程。Netflix 开发的 Chaos Monkey 工具可以随机关闭系统，这样可以测试整个监控、通知、分析和恢复流程。如果尚未这样做，可以将其作为运维团队定期执行的流程清单中的一部分，以确保团队成员都熟悉流程。这些活动不仅能够测试全量恢复和增量恢复，还能够将恢复融入复制流中，以将节点加入服务。

2. 构建不同的环境

为开发、集成测试、运维测试和演示构建环境不可避免。其中一些环境需要恢复完整的数据，这需要利用节点和整个集群的恢复能力。有些环境还有其他需求，比如用于功能测试的数据子集恢复，以及出于保护用户隐私目的的数据脱敏，这样就可以测试时间点恢复以及特定对象的恢复。这些都与标准的全量数据集恢复非常不同，对于运维人员或应用程序损坏时恢复数据非常有用。通过创建允许对象级和特定时间点恢复的 API，可以将这类流程自动化，提高熟练度。

3. 提取、转换和加载以及下游数据存储流水线

与构建环境类似，把数据从生产数据库导入流水线以进行下游分析和流式数据存储，这个过程非常适合采用特定时间点恢复流程、对象级恢复流程和 API。

4. 运维测试

在各种测试场景中，都需要数据副本。一些测试（比如容量测试和负载测试）需要完整的数据集，适于采用全量恢复流程。功能测试可能需要较小的数据集，适于采用特定时间点和对象级恢复。

恢复测试本身可以成为一个连续操作。除了在日常场景中利用恢复流程，还可以将恢复设置为持续运行，这允许进行自动测试和验证，以尽早发现可能会破坏备份过程的任何问题。当提到这个过程时，很多人会问如何验证恢复是否成功。

备份时可以生成测试所用的大量数据，例如：

- 自动递增的最新 ID；
- 对象的行计数；
- 只能插入的（因此可以视为不可变的）数据子集的校验和；
- schema 定义文件的校验和。

与任何测试一样，验证备份是否成功的测试应该分层进行。有些测试会很快成功或失败，这应该是测试的第一层，例如元数据 / 对象定义的校验和对比，数据库实例的启动，以及成功连接到复制线程。有些操作可能需要更长时间，比如数据校验和以及表计数，应该在稍后的过程中验证。

7.3.2 计划外的恢复场景

有了所有可以使用的日常计划的场景，恢复流程应该进行了很好的调优、撰写了详细的文档、进行了良好的实践，并保证基本没有 bug 和问题。因此，计划外的场景很少非常可怕。在计划外的场景练习中，团队应该看不到任何区别。下面列出每种计划外的场景，并就每种可能需要演练恢复流程的情况做深入讨论：

- 人为错误；
- 应用程序错误；
- 基础设施服务；
- 操作系统和硬件错误；
- 硬件故障；
- 数据中心故障。

1. 人为错误

理想情况下，很少发生人为错误。如果你正在帮工程师构建防御，应该可以添加很多预防措施。不过，总会有因运维人员误操作而造成损害的情况发生。例如在数据库上执行更新或删除时，缺少 `WHERE` 子句；或者在生产环境中执行了数据清理脚本，而不是在测试环境中；或者执行了正确的脚本，但是时间不对；又或者在错误的主机上执行了脚本。这些错误通常会被立即发现和恢复。然而，在某些情况下，这些变化的影响可能在几天或几周内都无法得知，因而无法有效地检测出来。

2. 应用程序错误

在讨论的所有场景中，应用程序错误是最可怕的，因为它们可能非常危险。应用程序不断修改与数据存储的交互方式，其中许多应用程序还管理引用完整性，以及指向文件或第三方 ID 等资源的外部指针。引入破坏性的改动数据、删除数据，或以在相当长一段时间内都不会被注意到的方式添加错误数据的变更，对应用程序而言是非常容易的。

3. 基础设施服务

第 6 章介绍了基础设施管理服务的强大。然而这些系统的破坏性可能不亚于其作用，编辑文件、指向不同的环境或推送不正确的配置，都会导致严重的后果。

4. 操作系统和硬件错误

操作系统及其所连接的硬件都是由人构建的系统，因此可能会由于没有记录文档或鲜为人知的配置而导致错误和意外后果。在数据恢复的上下文中，从数据库到操作系统缓存、文件系统和控制器，再到磁盘都是如此。数据损坏或数据丢失，比我们想象的要普遍得多。然而我们对这些机制的信任和依赖营造了一种文化，在这种文化中，人们相信数据的完整性而不是持怀疑态度。



静默损坏

2008 年，Netflix 曾经发生过这种操作系统和硬件错误。磁盘的错误检测和纠正机制采用 ECC（error correction code）。ECC 可以自动纠正 1 bit 的错误，并且能够检测到 2 bit 的错误。ECC 可以检测到的错误的汉明距离是可纠正错误的两倍。因此，在 512 字节扇区的硬盘中，如果可以纠正 46 字节的错误，则它最多可以检测到 92 字节的错误。无法纠正的错误会报告给磁盘控制器，磁盘控制器将增加 S.M.A.R.T 中“无法纠正错误”的计数；但多于 92 字节的错误会被视为正常数据而直接传递给控制器，并扩散到备份数据中，这可能会酿成大祸！

这就是为什么所谓的“云”或者“Serverless”计算值得高度怀疑。如果不了解实现细节，就无法确认数据完整性具有最高优先级。数据完整性的重要性通常被低估，甚至因性能原因而被忽略。无知便无力！

校验和文件系统（比如 ZFS）会对每个磁盘块做校验，探查损坏的数据。如果使用涉及镜像或奇偶校验的 RAID，甚至可以修复数据。

5. 硬件故障

硬件组件可能会发生故障，这在分布式系统中经常发生。磁盘、内存、CPU、控制器以及网络设备经常发生故障。这些硬件故障会导致节点失效，或节点延时增加进而导致系统不可用。共享系统中的设备（比如网络设备）故障会影响整个集群，导致集群不可用；或使整个集群分裂成较小的集群，而小集群并不知道网络已经分区，因此可能会快速导致数据出现较大差异，需要修复。

6. 数据中心故障

有时，网络硬件故障会引发级联故障，导致整个数据中心发生故障。存储系统的背板拥塞偶尔也会导致级联故障，例如 2012 年 AWS 发生的故障。有时，飓风、地震、交通事故等都可能导致整个数据中心发生故障。在这种情况下恢复系统，即使对最稳健的恢复策略而言也是一种考验。

7.3.3 场景的范围

列举了需要恢复数据的计划内的场景和计划外的场景之后，接下来考虑事件的范围，以便确定合适的应对方案。考虑以下几个方面：

- 局部或单节点；
- 集群范围；
- 数据中心或多个集群。

在**局部或单节点范围**内，恢复限于单个节点。比如向集群中添加新节点以扩容或替换故障节点，又或者进行滚动升级，逐个节点进行恢复，这些都属于**局部范围**。

在**整个集群范围**内，集群中的所有节点都需要恢复。比如数据发生破坏性改变或者删除时，这种数据损坏会通过复制扩散到所有节点。或许需要构建一个新集群以测试容量。

数据中心或多个集群范围是指需要恢复一个物理位置或者区域的所有数据。这可能是因为共享的存储系统发生故障，或者某种灾害导致数据中心发生灾难性故障，也可能因为按计划部署一个新的冗余区域。

除了按地域划分范围，还可以按数据集把范围分为以下 3 类：

- 单个数据对象；
- 多个数据对象；
- 数据库元数据。

单个数据对象范围指需要恢复某个数据对象的部分或全部数据。如前所述，DELETE 操作删除的数据比预期多导致的事故，就属于单个数据对象范围。**多个数据对象范围**是指某个数据库中多个（可能所有）数据对象的数据需要恢复。**元数据范围**是指存储在数据库中的数据是完好的，但丢失了使数据库可用的元数据，比如用户信息、安全权限信息、与操作系统文件的映射信息等。

7.3.4 不同场景的影响

除了定义需要恢复数据的场景，以及列举不同场景的恢复范围，定义潜在影响也很重要，以便选择恢复方案。对于不影响 SLO 的数据丢失，可以有条不紊地处理，逐步减轻数据丢

失造成的影响。对于会违背 SLO、更具破坏性的变更，在进行任何长期清理之前，必须考虑分类和快速恢复服务。可以将恢复方案按影响分为以下 3 类：

- 影响 SLO，应用程序退出，或者大部分用户受影响；
- SLO 很可能受影响，部分用户受影响；
- 只影响功能，不影响 SLO。

基于恢复场景、范围和产生的影响，有 72 种可能的场景组合需要考虑。这实在太多了，无法面面俱到。好在很多场景可以采用相同的恢复方案。即使有这种重叠，也无法为所有可能的情况都做周详的计划。因此，必须建立多层次的恢复方案，以确保有足够的恢复工具。下面利用本节所讲内容定义恢复策略。

7.4 恢复策略分解

称其为“恢复策略”而不是“备份策略”是有原因的。数据恢复是进行备份的原因，备份只是达到最终目的的手段，因此它取决于真正的需求：以给定参数恢复。对于“数据库有备份吗”这个简单问题，应该这样回答：“有，可以按多种方式恢复，具体取决于恢复场景。”只是简单给出肯定回答是轻率的，而且传递了一种错误的安全感，这种行为不负责任且危险。

有效的数据库恢复策略，不但需要以有效的方式应对多种场景，也包括数据丢失 / 损坏检测、恢复测试以及恢复有效性验证。

7.4.1 策略第1步：检测

尽早发现可能的数据丢失或者损坏至关重要。对于 7.3.2 节讨论的人为错误和应用程序错误，当发现这些问题时，它们可能已经持续了几天、几周甚至更久。这意味着，在意识到数据需要备份时，备份可能已经晚了。因此，对于所有工程而言检测都是高优先级的。除了尽早检测数据丢失和损坏，在早期检测失败的情况下，确保有尽可能大的恢复窗口也非常重要。下面探究之前讨论过的不同故障场景，以及检测和扩大恢复窗口的可行方案。

1. 人为错误

缩短发现数据丢失的时间的一个重要影响因素是，在生产环境中不允许执行手动变更或临时变更。创建脚本包装器或者 API 级别的抽象，可以指引工程师通过有效的步骤来确保所有变更尽可能安全、经过测试、文档完善并推广到相应的团队。

有效的脚本包装器或 API 可以：

- 通过参数化在多个环境中执行；
- 在演练阶段评估并验证运行结果；

- 包含代码执行状况的测试套件；
- 在执行后验证变更是否符合预期；
- 通过相同的 API 进行软删除或轻松回滚；
- 按 ID 记录所有修改的数据，以识别变更和恢复数据。

移除过程中的临时操作和手动操作，便于调试工程师追踪所有变更。所有变更都将被记录下来以便追溯，不会淹没在日常活动中。最后，通过对修改或删除等操作引入中间阶段，以及构建数据的轻松回滚，可以为发现和修正变更导致的问题赢得更大的时间窗口。但这并不能保证！虽然手动流程可能记录得非常好，但人们可能会忘记在自动化流程中设置日志，或者会绕过日志。

2. 应用程序错误

尽早检测到应用程序错误的关键是数据验证。当软件工程师引入了新对象或属性时，数据库可靠性工程师需要与他们合作，确定可以在应用程序之外的下游进行的数据验证。

和所有测试一样，初始工作应集中于快速测试。快速测试可为关键数据组件提供快速反馈，例如文件的外部指针、用于保证引用完整性的关系映射，以及 PII（personal identification information，个人身份信息）。随着数据和应用程序的增长，这项验证的成本会更高，也更有价值。应当营造一种文化，让工程师对数据质量和完整性负责，而不是对存储引擎负责，这既能保证使用不同数据库的灵活性，又能让人们对实验和快速开发应用程序功能更有信心。数据验证是一道防护栏，让大家更勇敢也更有信心。

3. 基础设施服务

任何对基础设施具有灾难性影响、需要恢复的事故，都应能通过监控系统迅速发现。话虽如此，还是有些变更不易察觉，但可能会造成数据丢失或损坏，影响甚至破坏可用性。使用黄金镜像并将其和基础设施中的组件定期进行对比，有助于快速发现基础设施与测试镜像之间的差异。类似地，使用版本控制的基础设施有助于发现基础设施的差异，以便向相关工程师报警或者自动修复。

4. 操作系统和硬件错误

与基础设施服务一样，监控日志和度量值可以快速捕获大部分问题。对于非标准的边缘场景，需要经过一些思考和拥有相关经验才能发现错误，并将它添加到监控中以便尽早检测到，磁盘块校验和就是一个例子。并不是所有文件系统都会做校验和检查，对于重要数据，团队需要仔细选择合适的能通过校验和发现静态错误的文件系统。

5. 硬件与数据中心故障

与基础设施服务一样，利用第 4 章所讲的监控很容易发现这些故障。我们已经做到了，很棒吧？

7.4.2　策略第2步：分层存储

有效的恢复策略依赖数据存放在多个存储层；不同的存储层可以满足不同的恢复需求，这不仅保证了合理的性能，也保证了不同场景下适当的成本和持续性。

1. 在线高性能存储

大部分数据存储产品在这种存储池上运行，其特点是高吞吐量、低延时，因此价格也较高。当恢复时间非常重要时，将数据库近期的备份以及相关增量备份放在该层非常关键。通常而言，只有近期数据的少量备份会驻留于此，以便能在最常见和影响最大的场景中快速恢复。典型例子有：在服务发生故障后，将整个数据库复制到新加入生产环境中的节点，或者因流量快速增加而需要扩容的情况。

2. 在线低性能存储

这种存储池通常用于存储对延时不敏感的数据。这类存储池通常由吞吐量低、对延时不敏感、廉价的大容量磁盘组成。因其容量较大，可以将需要保存较长时间的备份存储在该层。相对不频繁、影响较小或者运行时间较长的备份场景，可以使用这些较早的备份。典型例子是寻找并修复早期检测遗漏的应用程序错误或人为错误。

3. 离线存储

离线存储指磁带存储或者类似于 Amazon Glacier 这样的存储。这类存储是异地的，通常需要车辆将其运到需要恢复数据的区域。它可以满足业务连续性和审计要求，但不能用于日常的恢复。但是，出于容量和费用方面的考虑，这类存储的可用存储空间较大，可以存储企业生命期内或者至少合规期限内的所有数据。

4. 对象存储

对象存储是一种将数据作为对象而非文件或块进行管理的存储系统。对象存储提供了传统存储系统所不具备的功能，例如应用程序可用的 API、对象的多版本以及基于复制机制和分布式的高可用性。对象存储系统使大量具有完整版本和历史的对象具有可用性、扩展性和自修复性。这对于轻松恢复非结构化、不依赖与其他数据的关系来获得一致性的对象是理想的选择。此外，它也有助于从应用程序错误或人为错误中恢复。Amazon S3 是廉价、可扩展且可靠的对象存储的典型。

这些层中的每一层，都是跨场景可恢复性综合策略中的一部分。因为我们无法预测所有可能的场景，所以每一层都是必需的。接下来讨论利用这些存储层提供可恢复性的工具。

7.4.3　策略第3步：多样的工具集

下面基于场景和评估选项来评估所需的恢复流程。前面介绍了一系列可用工具，接下来仔细了解这些工具。

副本不是备份

本书不把副本视为一种有效地备份数据以便进行恢复的方法。复制过程是盲目的，可能会扩散人为错误、应用程序错误和数据损坏。应将副本视为数据移动和同步的必要工具，而不是用于数据恢复的工具。如果有人说，他们使用副本恢复数据，请投以鄙视的目光。同理，RAID 也不是备份，而是冗余。

1. 全量物理备份

需要对所有级别进行全量恢复：节点级别、集群级别和数据中心级别。快速、可移植的全量恢复非常强大，在动态环境中是必需的。全量备份可以快速构建节点以扩容，或在故障期间替换故障节点。可以通过网络或通过与对应主机 / 实例直接连接的方式进行全量备份。要想执行全量恢复，就需要全量备份。

关系型数据库的全量备份需要锁定数据库以获得可以复制的一致快照，或者能够在复制期间关闭数据库。在异步复制的环境中，不能保证副本与主库写入是同步的，因此，应尽可能从主库执行全量备份。在创建数据库快照、文件系统或基础设施的快照之后，应将快照复制到其他存储系统中。

对只追加写型数据库（比如 Cassandra）的全量备份，涉及在操作系统级别利用操作系统的硬链接特性做快照。由于这类分布式数据存储中的单个节点并没有全量数据，因此该备份被视为**最终一致的**。恢复数据需要将节点重新加入集群，此时系统内部的同步机制最终使节点达到最新状态。

线上高性能存储的全量备份可快速切换为线上集群。这类备份通常是未压缩的，因为解压缩很耗时。线上低性能存储上的全量备份用于构建不同的环境，例如用于测试、分析或数据取证。压缩是在有限的存储空间内更长时间地保留全量备份的一种有效方式。

2. 增量物理备份

如前所述，增量备份可以弥合上次全量备份与其后某个时间点之间的差距。增量物理备份通常包含已更改的数据块。无论从备份还是存储的性能影响方面来看，全量备份的开销都很大；但通过增量备份可以快速地基于之前的全量备份达到最新状态，以便在集群中正常使用。

3. 全量逻辑备份与增量逻辑备份

全量逻辑备份具有可移植性，并简化了数据子集的提取。这种备份不用于节点的快速恢复，而用于取证、在节点之间移动数据以及恢复大数据集中的特定数据子集。

4. 对象存储

与逻辑备份一样，对象存储可以轻松恢复指定对象。事实上，对象存储针对此用例做了优化，并且可以根据需要以 API 编程的方式轻松恢复对象。

7.4.4 策略第4步：测试

对于像恢复这样重要的基础设施流程，令人惊讶的是测试常常被抛在一边，而测试是确保备份能正常用于恢复的重要手段。关于恢复的测试往往是间歇性的（例如每月或每季度一次）。尽管聊胜于无，但两次测试之间有较长的时间间隔，其间备份过程可能会停止工作。

将测试添加到正在进行的流程中有两种有效的方法。第 1 种方法是将恢复纳入日常流程。这样，可以持续对恢复进行测试，从而快速识别错误和故障。此外，持续恢复可以获得恢复用时的相关数据，这对于校准恢复流程以符合 SLA 至关重要。将恢复不断集成到日常流程中的示例包括：

- 构建集成环境；
- 构建测试环境；
- 定期替换生产集群中的节点。

如果环境中无法重建数据存储，也可以创建一个持续的测试过程，从而持续地恢复最近的备份，进而验证恢复是否成功。无论测试过程是否已经自动化，都要偶尔测试异地备份。

这些步骤可以为不同的恢复场景提供可靠的保障。根据场景和工具，可以评估开发和资源方面的需求。

7.5 既定恢复策略

如前所述，需要准备好应对多种故障场景。为此，需要一个丰富的工具集以及使用这些工具的计划。

7.5.1 在线快速存储的全量备份和增量备份

该策略是日常恢复中最基础、最核心的部分。当生产环境或测试环境需要快速添加新节点时，可以采用该策略。

1. 用例

该策略的主要应用场景如下：

- 更换故障节点；
- 添加新节点；
- 构建用于功能集成的测试环境；
- 构建用于运维测试的测试环境。

由于备份期间的延迟较大，因此全量备份通常最多每天进行一次。只要保留一周的数据，就可以快速访问最近的任何更改。这涉及数据库未压缩的 7 个全量备份，以及增量备份中的所有变更。某些环境没有足够的容量或预算，因此可能会在保留时长和备份频率间进行权衡。

2. 检测

当节点或组件发生故障需要恢复时，监控能发出通知。通过容量规划评审和预测可以判断何时需要添加更多节点来扩容。

3. 分层存储

由于生产环境中的故障需要快速恢复，因此需要高性能的在线存储。同样，测试速度必须尽可能跟上开发速度。

4. 工具箱

全量物理备份和增量物理备份是最高效的恢复方案，因而在此处最为合适。因为恢复时间的要求，所以这些备份未被压缩。

5. 测试

因为经常进行集成测试，所以会经常遇到此类恢复场景。在虚拟环境中，每天在集群中添加一个新节点，即可每天执行恢复流程。此外，鉴于该流程的重要性，因而引入了持续恢复流程。

7.5.2 在线慢速存储的全量备份和增量备份

这里讨论的慢速存储，访问速度较慢，更便宜，存储空间也更大。

1. 用例

该策略的主要应用场景如下：

- 应用程序错误；
- 人为错误；
- 数据损坏修复；
- 构建运维测试环境。

当因上线新功能、变更失败或迁移不当导致数据损坏时，就需要能够访问并提取大量数据用于恢复。这种场景中的恢复流程最为混乱，因为要考虑各种潜在问题。在恢复期间，通常需要编写脚本，如果没有进行有效的测试，这些脚本自身可能会导致更多 bug 和错误。

将全量备份压缩后从高性能存储复制到低性能存储，是当前策略下获取全量备份的简单方法。由于压缩和更便宜的存储，根据预算和需求保留一个月甚至更长时间变得可能。在快速变更的环境中，出现数据损坏和完整性问题的概率要高得多，这意味着备份和恢复更耗时。

2. 检测

数据验证是决定是否需要恢复的关键。当验证失败时，工程师可以使用这些备份来查明发生了什么、何时发生的，然后提取正确的数据来修复生产环境。

3. 分层存储

由于该策略需要保存备份很长时间，因此需要低性能的在线存储。廉价的大容量存储是不错的选择。

4. 工具箱

在这种情况下，全量物理备份和增量物理备份最为合适。由于对恢复时间要求不高，因而此类备份也会被压缩。除了物理备份，还可以利用逻辑备份（例如复制日志）来提高恢复的灵活性。

5. 测试

由于这种恢复不常发生，因此持续的自动恢复流程对于确保所有备份都可用且状态良好至关重要。偶尔演练特定恢复场景（例如一个表或一个数据范围）有助于团队熟悉恢复流程和工具。

7.5.3 离线存储

到目前为止，这种方式最便宜，提取数据也最慢。

1. 用例

该策略的主要应用场景如下：

- 审计与合规性；
- 业务连续性。

实际上，这种方式专注于罕见但关键的需求。审计与合规性通常需要追溯 7 年甚至更久之前的数据，但是它们对时间不敏感，可能需要相当长一段时间来准备和展示。业务连续性要求数据副本远离当前生产系统的物理位置，以确保在发生灾难时可以重建。尽管在这种情况下时间紧急，但可以采用灵活的方式来恢复。

将全量备份压缩后从低性能存储复制到离线存储，是当前策略下获取全量备份的简单方法。备份保留 7 年甚至更长时间不仅是可能的，而且是必需的。

2. 检测

检测在该策略中并不重要。

3. 分层存储

由于该策略需要保存备份很长时间，因此需要廉价的大容量存储，磁带或 Amazon Glacier 之类的解决方案通常是不错的选择。

4. 工具箱

全量备份在这里最合适。由于对恢复时间不敏感，因此这些备份也会被压缩。

5. 测试

这里的测试策略类似于在线慢速存储层。

7.5.4 对象存储

对象存储的一个示例是 Amazon S3。对象存储的特点是通过编程访问，而不是通过物理访问。

1. 用例

该策略的主要应用场景如下：

- 应用程序错误；
- 人为错误；
- 数据损坏修复。

软件工程师可将对象存储查看、存放和获取 API 集成到应用程序和管理工具中，以便有效地从人为错误和应用程序错误中恢复。使用版本控制，管理员可以轻松地从删除、意外修改和其他类似情况中恢复，而无须使用这些工具。

2. 检测

数据验证是决定是否需要恢复的关键。当验证失败时，工程师可以确定事故发生的时间范围，并通过编程来恢复。

3. 测试

由于对象级别恢复是应用程序的一部分，因此标准集成测试应确保其有效。

这 4 种数据恢复方法能为大多数场景下（甚至是意料和计划外）的数据恢复提供相当全面的策略。根据恢复服务等级的目标、预算和资源，还需要进行微调。但总体而言，我们已为制定有效的数据恢复计划奠定了基础，该计划将检测、度量值、追踪和持续测试结合在了一起。

7.6 小结

本章详细探讨了可能导致需要进行数据恢复的潜在风险，这样的风险很多且不可预测。重要的一点是我们无法为所有事情做计划，因而需要制定全面的策略，以确保可以应对可能发生的任何事情。其中一些工作包括与软件工程师合作，将恢复流程融入应用程序。有些情况下，需要自己构建一些可靠的恢复软件。而且，在所有情况下，都必须在前几章内容（有关服务等级管理、风险管理、基础设施管理和基础设施工程）的基础上进行规划。

第 8 章将讨论发布管理。在阅读本书其余部分时，请始终将数据恢复牢记于心。应用程序和基础设施每前进一步，都会给数据和有状态的服务带来风险。数据库可靠性工程师的首要职责是确保数据可恢复。

第 8 章

发布管理

随着推行自动化，管理基础设施的负担减轻了，数据库可靠性工程师能够专注于更有价值的工作，包括与软件工程师合作，一起构建、测试和部署应用程序功能。传统上，数据库管理员是生产环境的“守卫”。他们希望了解数据库迁移、数据库对象定义以及访问数据库的代码，以确保相关操作能正确执行。当满足要求时，数据库管理员会进行适当的手动变更并将其投入生产。

你可能会想，对于数据库结构经历大量部署和变更的环境来说，该过程不一定是可持续的。实际上，如果你亲自参与部分流程，就会清楚地感受到数据库管理员从“守卫”变成“瓶颈”的速度有多快，这会导致数据库管理员心生倦怠和软件工程遭遇挫败。

本章主要探讨数据库可靠性工程师如何有效地利用时间、技能和经验来支持软件工程（采用持续集成甚至持续部署），避免成为软件工程的瓶颈。

8.1　培训与合作

数据库可靠性工程师必须首先对开发人员进行培训，让他们了解正在使用的数据存储。如果软件工程师能够对数据结构、SQL 和整体交互策略做出更好的选择，那么需要数据库可靠性工程师直接干预的情况将减少。通过向软件工程师团队传授数据库知识，数据库可靠性工程师会对同事的持续学习过程产生重大影响。这也会增进彼此的关系、信任和沟通，对于技术组织的成功至关重要。

需要明确的是，数据库可靠性工程师不应对软件工程团队不闻不问，双方应增强互动。数据库可靠性工程师通过常规的互动方式和战略性努力来创建一个专业的团队：可以轻松获

取相关资源，并在数据库相关的日常工作中自主决策。

切记，所做的一切要具体化、可测量且可操作。为团队的成功定义关键指标，并在实施策略和变更时观察它们如何为团队提供帮助。在此过程中要考虑如下关键指标。

- 需要数据库可靠性工程师介入的数据库相关任务的数量。
- 成功 / 失败的数据库部署。
- 新功能上线的速度。软件工程师能以多快的速度将新功能上线？
- 由数据库变更导致的宕机时间。

敏捷方法论和 DevOps 文化，需要具有不同背景、技能水平和专业知识的人跨职能互动，以便紧密协作。培训和协作是该过程的重要组成部分，也是数据库可靠性工程师从传统的“数据库管理员”模式转变成技术团队重要组成部分的绝佳机会。

8.1.1 收集并分享相关资讯

毫无疑问，应该关注数据和数据库领域内技术专家或优秀组织的博客、Twitter 话题以及社交账号。在此过程中，你会发现与你和团队正在做的事情相关且有价值的文章、问答、播客和项目，可供研究和分享。创建简讯、论坛甚至聊天频道，在其中定期发布相关信息并进行讨论。向工程团队展示数据库可靠性工程师在帮助他们取得成功和持续开发方面所付出的努力。

8.1.2 促进对话

下一步是与软件工程师积极地沟通和互动。此时，你和团队将深挖分享过的相关内容以激发灵感，学习利用这些信息，甚至找出团队之间在理解上的偏差，并一起研究和实验以便改进系统。实现这一点有多种方式，具体主要取决于所在环境的学习和协作文化。以下是几种方式：

- 每周技术讲座；
- 午餐期间的简短交流；
- 在线问答；
- 专注于知识分享的聊天频道。

此外，可以设置开放交流时间，鼓励大家提问，就特定话题进行交流和探讨。

8.1.3 特定领域知识

前面介绍了与数据存储和架构相关的基础知识，下面介绍特定领域的相关知识。

1. 架构

我们不推崇静态文档，因为不能与实际构建和部署架构的流程相结合。通过配置管理和编排系统，可以获得大量最新文档。基于此构建工具，以方便查找、借阅和评注文档，可以为团队创建实用的文档。

更重要的是要理解上下文和历史背景。使用特定的数据存储、配置和拓扑结构是有原因的。帮助工程师搞清楚一些问题，包括：正在使用的架构是什么？为什么要使用这些架构？如何获得与架构的使用相关的文档？做出了哪些权衡和折中才达到了现在的状态？

数据库可靠性工程师需要向工程师提供这些知识、上下文和历史背景。这样一来，无须指导，工程师也能在日常开发中独立决策。建立设计文档的知识库，是构建架构上下文和历史背景的前提。这些文档可以用于需要新架构组件的整个项目、增量变更或子项目。例如，你肯定需要一个设计文档来记录从基于语句的复制到基于行复制的演变过程，但它的要求不一定与首次安装 Kafka（以便为事件驱动的架构创建分布式日志系统）时相同。

创建和分享文档模板是一项团队活动，重要的是要包括如下信息。

- **执行摘要**

  给那些查找基础知识的人。

- **目标和非目标**

  这个项目的预期结果是什么？什么超出了范围？

- **背景**

  将来的读者可能需要的上下文。

- **设计**

  从上层到具体细节，应该包含图表、样本配置或算法。

- **约束**

  需要牢记要解决什么问题。例如对 PCI 的合规性、IaaS（Internet as a service，网络即服务）的特定需求或人员配备。

- **备选方案**

  是否评估了其他选择？基于什么方法论？为什么不选择它们？

- **启动详情**

  如何启动？出现了什么问题以及是如何解决的？应该包含脚本、流程和注释。

显而易见，这些文档可能会变得很大，对于某些项目来说这没有问题。分布式系统和具有多层的服务很复杂，包含很多信息和上下文。请记住，这里的重点是让工程师了解上下文，节省数据库可靠性工程师的时间。

2. 数据模型

数据流和物理通道（physical pipeline）是与所存储的数据种类有关的信息，它和架构同样重要。让软件工程师知道已存储数据的种类和位置，就能消除开发过程中不必要的调研。此外，数据库可靠性工程师可以分享在不同数据库（关系型、键值型或文档型）中表示相同数据的方式，也可以借机宣传最佳实践（哪些数据库不适合存储哪些类型的数据）。

3. 最佳实践和标准

为工程师定期进行的活动制定标准，是提升数据库可靠性工程师价值的另一种有效方式。数据库可靠性工程师可以在帮助工程师制定决策时逐步给出这些标准，例如：

- 数据类型标准；
- 索引；
- 元数据属性；
- 数据存储选型；
- 提供的度量值；
- 设计模式；
- 迁移和数据库变更模式。

在与工程师一起工作时发布这些信息，形成可以随时访问的自助服务知识库[1]，以避免自己成为团队的瓶颈。

4. 工具

为软件工程师提供高效的开发工具是终极赋能方式，例如使用基准测试工具和脚本、一致性验证程序、模板甚至新数据存储的配置程序来提供帮助。从根本上来说，这些措施旨在加快开发速度，同时腾出时间来做更有价值的工作。

优秀的工具有：

- Etsy 开发的 Schemanator 工具；
- Percona 工具包，尤其是在线 schema 变更；
- SQL 调整和优化套件；
- SeveralNines 的 Cluster Configurator 工具；
- 签入变更计划模板和示例；
- 签入迁移脚本和模式示例；
- 基准测试套件，以方便测试、可视化和分析。

把软件工程团队视为客户，并实施精益产品开发策略。为他们提供一个可行的最小工具集，并持续地关注、追踪并评估他们的成功、失败、痛点和需求，以了解哪些工具会为他们带来最大收益。

注 1：Martin Fowler. Evolutionary Database Design.

8.1.4 协作

如果定期组织培训、创建工具并赋予工程师权力，自然会建立良好的关系。这很关键，因为良好的关系会促进持续协作。任何软件工程师都应有权和数据库可靠性工程师团队联系，以查询信息或在工作期间寻求合作。当软件工程师了解数据库可靠性工程师团队的工作方式和目标，数据库可靠性工程师团队也了解软件开发的更多信息时，双方都会获益。

更进一步，数据库可靠性工程师可以主动联系软件工程师。有些场景对数据库开发和重构有很强的依赖，数据库可靠性工程师应当集中精力确保团队的效率并取得成功。在这些场景中，需要结对工作或成为团队的一部分。同样，密切关注即将签入代码主干的迁移，有助于数据库可靠性工程师团队轻松找到需要审查之处。

毫无疑问，确保数据库可靠性工程师不单打独斗或远离代码构建，将有助于确保这种协作顺利开展。将数据库可靠性工程师和软件工程师的项目和工作互换，也可以实现这一点。

前面一直在讨论，如何帮助软件工程师在开发过程中尽可能实现自给自足。随着开发团队的成长，需要利用有效的培训、标准和工具来确保团队可以正确决策，而无须数据库可靠性工程师直接干预。此外，在培训中告诉软件工程师，何时需要数据库可靠性工程师审查待定的变更和解决方案，以便在必要时提供帮助。

接下来讨论数据库可靠性工程师如何有效地支持交付流水线的各个组件。CD（continuous delivery，持续交付）并非新概念，但是各组织一直在努力将数据库纳入流程。下面讨论如何有效地将数据库层引入整个交付周期。

8.2 集成

频繁地集成数据库变更，会使得变更集更小、更易于管理，而且能通过快速速识别重大变更来对变更的影响迅速做出反馈。许多组织努力实现 CI（continuous integration，持续集成），以便自动集成提交的所有变更。CI 的很大一部分价值是自动化测试，用于检验数据库是否满足应用程序的需求。只要提交代码，就会触发这些自动化测试。

在整个软件开发生命周期中，对数据库代码或组件的任何变更都应触发全新的构建，然后进行集成和测试。数据库可靠性工程师和软件工程团队负责确定数据库正常运行的标准。在软件工程师重构数据模型、引入新数据集和使用新方式查询数据库时，集成测试能够再次验证数据库，以确保其工作状态是正常的。

事实证明，对数据库层进行 CI 颇具挑战。除了应用程序使用数据库对象的功能方面，在可用性、一致性、延时和安全性方面还有运维操作的要求。变更对象会影响已存储的代码（功能、触发器或视图等），甚至会影响应用程序其他部分的查询。此外，数据库中的高级功能（例如事件）会导致脆弱性加重。除了功能测试，还有许多潜在的涉及数据完整性的

极端情况。即使有时数据库约束可以强制保证完整性，也必须测试这些规则。更令人担忧的是，有的环境没有采用数据库级别约束。

先决条件

要在数据库级别建立 CI，必须满足以下 5 个要求。

1. VCS

与基础设施的代码和配置一样，所有数据库迁移都必须和应用程序的其余部分一起提交到同一个 VCS 中。至关重要的是能够根据最新配置进行构建，以了解最近的数据库配置变更如何以新的方式破坏应用程序构建。

为了防止出现不一致的情况，**所有**内容需要提交到代码库中，包括：

- 数据库对象迁移；
- 触发器；
- 存储过程和函数；
- 视图；
- 配置；
- 功能采样数据集；
- 数据清理脚本。

这在 CI 之外还带来了其他好处。

- 在一处就能轻松找到所有相关项。
- 支持自动化部署需要的一切组件（参见第 6 章）。
- 可以找到数据库的所有历史记录和版本，这有助于恢复、取证和故障排除。
- 应用程序和数据库版本将是同步的，至少在理想情况下如此。

当数据库可靠性工程师根据已知的工作状态，持续进行集成、验证签入的代码和基础设施更改时，软件工程师可以将最新的数据库版本应用于开发环境。

2. 数据库构建自动化

假设你正在使用第 6 章所讲的配置管理和自动化技术，那么在集成时应该能够自动构建数据库。这包括应用最新的 DDL（data definition language，数据定义语言）和脚本，以及加载用于测试的代表性数据集。这可能比预期的更具挑战性，因为必须经常清理生产数据或对其脱敏，以避免在暴露用户数据时出现合规性问题。

3. 测试数据

空数据库几乎总是表现得非常好，小数据集通常也是如此。这里需要 3 组数据。首先是查找表需要的所有元数据，其中包含用户类型、位置 ID、工作流和内部表的 ID。这类数据

集通常很小，但对于应用程序的正常工作而言至关重要。

其次需要功能数据，例如用户或订单。在投入更多时间进行密集测试之前，功能数据通常足以支持早期的快速测试。

最后需要大型数据集，以帮助理解生产负载下的情况。这通常需要从生产数据集构建，并擦除敏感信息，以避免意外暴露用户数据，或不小心向数千名用户发送电子邮件，或者引发其他用户问题和法律问题。

元数据和测试数据集应该签入，作为构建的一部分进行版本控制。较大的数据集通常来自生产环境，应该对恢复和清理数据所需的脚本进行版本控制和签入，以确保应用程序和持久层之间同步。

4. 数据库变更和打包

一切都是以把实施数据库变更视为迁移（增量代码变更）为前提的。每一组变更，例如 alter 表、添加元数据或新列族，都将签入并获得一个序列号。因为所有变更都是按顺序实施的，所以任何时候都有一个与最近的变更对应的版本。

传统上，数据库管理员要么从开发人员那里获得变更列表，要么在开发和生产之间生成 schema diff，以获得为了发布应用必要变更所需的信息。这样做的好处是，数据库专家可以谨慎地管理那些影响可能很大的变更。在复杂的迁移期间，这样做可以将潜在的宕机时间及其影响最小化。

然而，这种传统方法的缺点是，很难看出哪些数据库更改对应哪些功能。如果必须回滚某些内容，那么识别与特定功能相关的数据库增量变更会很有挑战性。类似地，如果数据库变更失败，所有等待这些变更的功能都将被延迟，从而影响相应功能部署到生产环境的时间。

采用增量方式可以实现敏捷方法所能实现的一切：快速投入市场、较小的增量式变更、清晰的可见性，以及快速的反馈循环。但这意味着软件工程师必须了解如何安全地进行数据库 schema 变更，以及何时应该向数据库可靠性工程师团队求助。此外，还可能存在 schema 变更发生冲突的风险。如果两名软件工程师正在修改相同的对象，将导致重复进行此类变更。如果该数据库对象中有大量数据，schema 变更的用时将大增。在这种情况下，必须权衡利弊，软件工程师必须意识到相互冲突的可能。

5. CI 服务器和测试框架

假设你的软件集成已经运用了上述知识。一个好的 CI 系统将提供集成所需的所有功能，而测试框架将提供系统级测试和代码组件测试。

在系统级别，可以使用 Pester（Windows）或 Robot（Linux）之类的框架。也可以使用 Jepsen，该分布式系统测试框架专门用于验证分布式存储中数据的一致性和安全性。

有了这些先决条件，就可以开始使用公司的 CI 平台进行数据库 schema 变更了。顾名思义，CI 意味着只要签入数据库变更，就会自动进行集成测试。为了实现这一目标，并让工程团队确信这些变更不会对应用程序的预期功能和服务等级产生不利影响，测试就成了关键工具。

8.3 测试

所有工程师都向 VCS 签入数据库变更。CI 服务器能够自动触发数据库构建，并且构建的版本与应用程序版本一致，而且有一个测试框架。接下来做什么？需要验证集成是否有效，以及在下一个阶段（部署）中会产生什么影响。

坦白说，这很困难。数据库变更因会影响大量代码和功能而“臭名昭著”。话虽如此，有一些方法可以简化应用程序构建。

8.3.1 测试友好的开发实践

在设计开发流程时，可以通过多种方式来简化测试，下面介绍两个例子。

1. 抽象和封装

有许多方法可以帮助软件工程师抽象数据库访问。为什么要这么做？把数据库访问代码集中在一起，创建了一种标准的、易于理解的方式来实现新对象或者访问已有对象。这也意味着在变更数据库时，无须搜索整个代码库。这极大地简化了测试和集成。实现这种抽象的方式包括：

- DAO（data access object，数据访问对象）；
- API 或 Web 服务；
- 存储过程；
- 专门的框架。

有了这些，集成就可以首先集中测试访问和更新数据的代码，检查变更是否影响了功能。与任何测试一样，首先需要测试影响大、执行快的用例，而数据访问代码的集中有利于实现这一点。

2. 追求高效

工程师常用“select *”获取所处理对象的一整行。这样做是为了“防患于未然”，或者确保在任何需要相关数据的地方都能使用数据。也许他们想确定，如果在对象中添加一个属性，就会自动获取它。这种行为很危险并且是一种浪费，还会在变更期间将应用程序置于险境。“select *”将获取所有列，但如果代码没有做相应处理，程序就可能崩溃。获取的所有数据必须通过网络传输，如果获取多行，就需要更多带宽，可能导致 TCP 拥塞。有选择性地获取相应的列格外重要。可以适时修改访问对象的代码，并为此做好准备。

8.3.2　变更签入后的测试

代码签入后的测试旨在验证变更是否成功，以及应用程序是否工作正常。此外，可以在这个层面进行针对安全性和合规性的影响分析和基于规则的验证。签入代码之后，构建服务器应该立即构建一个用于集成的数据存储，然后实施变更并开展一系列测试。这些测试的运行速度很快，可以将结果迅速反馈给工程师。这意味着利用签入的最小数据集快速构建数据库，该数据集包含所有必要的元数据、用户账户数据和测试数据，这些测试数据是验证所有 DAO 功能是否正常所必需的。这样工程师就可以快速验证变更是否破坏了构建。

在组织的早期，可能这些工作大部分是手动完成的。当开始应用规则时，可以利用工具和自动化使这些过程更快、更可靠。

1. 构建前

在实施变更前，可以针对已确立的影响分析和合规性规则进行以下验证。

- 验证 SQL 语句是否正确。
- 验证可能受变更影响的行数。
- 验证为新列创建的索引。
- 验证表中有数据的新列不采用默认值。
- 验证对保存的代码和引用约束的影响。
- 报告正在更新的敏感数据库对象和属性。
- 当违反合规性时进行报告。

2. 构建

运行构建时将再次验证 SQL 语句。在这种情况下，是基于变更的实际应用，而不是基于规则的分析。

3. 构建后

对构建实施变更之后，可以运行功能测试套件。还可以创建报告，以分析变更产生的影响以及任何违规行为。

8.3.3　完整的数据集测试

应用程序在完整的生产数据集上运行后，服务可能无法达到预期的服务等级，这意味着测试套件应该在适当的负载下在生产数据集上运行。这需要更多准备工作和资源，因此可以将这类测试套件从标准的签入集成测试中剥离出来，异步地调度。根据集成和代码签入的频率，每周甚至每天执行此类测试可能更有意义。

这种覆盖面广的测试所采取的步骤各不相同，但通常遵循如下方式：

- 创建数据存储和应用程序实例；

- 部署代码；
- 恢复完整的数据集；
- 将数据匿名化；
- 关联度量集合；
- 实施数据存储的变更；
- 启动功能，快速测试；
- 执行负载测试，提高并发性；
- 销毁实例；
- 测试后分析。

在这些测试中，需要注意以下几点。

- 与之前在较小数据集上运行相比，延迟发生的变化。
- 数据库查询优化器中访问路径的改变，可能影响延迟或资源利用率。
- 数据库度量值，表明潜在的性能或功能影响（锁、错误、资源等待）。
- 与之前相比，资源利用率的变化。

可以将一些分析自动化，例如将查询注册到一个集中的数据存储中，并比较历史变更。有些度量分析需要运维人员进行有效的评审，以确定变更可否通过。

如果发现任何危险信号（自动化的或非自动化的），数据库可靠性工程师能够快速回顾自上次测试运行以来实施了哪些变更，以缩小分析范围。虽然不能立即确定危险是由哪个变更导致的，但有助于更快速地做出判断。

除了分析应用程序的响应速度，还必须定期对快速演变的数据存储执行其他测试。这些测试确保了数据库虽然不断变化，但在整个生态系统中一直表现良好。这些测试指下游测试和操作测试。

8.3.4 下游测试

下游测试用于确保数据存储的任何数据流水线和数据使用者不会受到实施数据库 schema 变更的不利影响。与完整的数据集测试一样，下游测试最好在签入过程中异步完成。下游测试的例子有：

- 验证由数据库中的数据所触发的事件工作流；
- 验证提取、转换，以及把数据加载到用于分析的数据存储中的过程；
- 验证直接与数据库交互的批处理和调度作业；
- 验证作业时间有无显著增加，时间的增加可能影响交付时间或下游流程。

与全量数据集测试类似，这些测试通常覆盖很多用例，并且需要使用更大的数据集。通过异步而有规律地运行这些测试，更容易识别影响了已在测试中标记的下游流程的潜在变

更。如果测试失败，可以停止上线；如果违反了规则，可以自动创建工单。

8.3.5 操作测试

随着数据集的增加和 schema 的演变，操作流程可能会运行更长时间，并且可能会发生故障。这些操作流程测试包括以下内容：

- 备份和恢复流程；
- 故障转移和集群操作；
- 基础设施配置和编排；
- 安全测试；
- 容量测试。

这些测试应该定期从生产数据集执行自动构建，并在运行测试之前应用所有挂起和签入的变更。失败的测试可以告知构建服务器，在把变更部署到生产环境之前，必须评估问题并予以解决。虽然数据库变更很少影响这些流程，但在服务等级上的影响可能很严重，因此需要高度关注。

通过结合持续、轻量级的构建和测试，以及策略性地进行更多测试，可靠性工程师、软件工程师、运维人员和项目管理方面将更有把握，即数据库变更可以部署到生产中，而无须数据库可靠性工程师团队直接干预。

此类集成是数据库可靠性工程师通过流程、知识共享和自动化提供价值的范例，使得软件工程师不受数据库可靠性工程师资源瓶颈的限制。下面讨论更重要的问题——部署。认识到数据库变更的安全性是第一步，而将这些变更安全地部署到生产环境中同样重要。

8.4 部署

前面讨论过数据库 schema 变更的概念及其优缺点。鉴于数据库变更的重要性，让软件工程师轻松、安全、增量式地在环境中迁移数据意义重大，至少要以尽可能安全的方式实施变更。

在理想的情况下，我们的目标是让软件工程师能够识别何时需要数据库可靠性工程师对其数据库变更进行分析和管理，以便有效地将变更部署到生产环境中。此外，我们会为软件工程师提供工具，让他们能够安全、稳妥地将大多数变更部署到生产环境中。最后，我们将使软件工程师能够随时将变更部署到生产环境中，而不是仅在维护期间。下面讨论如何实现这一点。

8.4.1 迁移和版本

如 8.2 节所讨论的，每个数据库变更集都应该有一个数字版本，通常是在实施变更集之后，

在数据库中存储一个递增的整数值。通过这种方式，部署系统可以轻松查看数据库 schema 的当前版本，进而简化实施变更。如果部署的代码对应数据库版本 456，而当前的数据库版本是 455，那么部署团队就知道，在部署代码之前必须应用 456 号变更集。

因此，软件工程师已将 456 号变更集签入代码库，并且成功运行了集成测试，所有功能都运行正常。接下来该做什么？

8.4.2 影响分析

前面讨论了提交后测试的影响分析。存在某些影响的变更（例如数据库中存储的代码无效或违反安全控制）是禁止发布的。软件工程师需要先修改其变更，直至消除这类影响。

下面介绍在生产系统上实施数据库变更的影响。

1. 锁定对象

许多变更会导致一个表甚至一组表不可写、不可读或者不可读写。在这种情况下，需要估算对象不可访问的总时长，并判断是否可接受。可接受的锁定时长是 SLO 或业务需求的一部分，因此是主观的。以前对这些对象实施的变更，可以用实施变更所需时间的特定度量来记录。这能提供一些客观数据来确定影响时间，但需要注意，变更所需时间会因数据集的增大以及操作的增多而延长。

如果影响时间不可接受，数据库可靠性工程师应该与部署团队共商计划，将时间缩减到可接受的范围，或者迁走线上流量，直到变更成功为止。

2. 资源饱和度

变更可能会消耗大量 I/O，也可能造成数据库其他事务的延时增加。这会导致违背服务等级，并最终导致进程积压，应用程序变得不可用，其他资源达到饱和。这很容易导致级联故障。

3. 数据完整性问题

变更过程中可能会有过渡期，其间约束会放松或推迟。同样，锁定和验证失效会导致数据以意外方式保存。

4. 复制停顿

数据库更改可能导致资源消耗增加和复制滞后，这可能影响副本的有效性，甚至将故障转移置于险境。

数据库可靠性工程师需要帮助软件工程师提前识别并避免这些影响。

8.4.3 变更模式

分析完影响后，软件工程师可以决定合适的变更部署方式。很多变更场景不必进行大量的增量变更和评审，新增对象、插入数据以及其他类似操作可以轻松地发布到生产环境中。

然而，在数据存入系统后，如果更改或删除这些数据以及更改或删除包含数据的对象，这类变更很可能会影响服务等级，此时软件工程师可以向数据库可靠性工程师求助。数据库可靠性工程师可以计划相对有限的一组变更。当和软件工程师一起计划并执行这些变更时，数据库可靠性工程师可以为将要使用的数据库更改创建一个模式库。如果这些模式经常执行且对服务无影响，可以将它们自动化。

例如，可以在集成和测试中建立部署准入规则，利用基于规则的分析和测试结果来判断部署变更是否安全。典型的操作包括如下内容。

- 更新和删除不用 `WHERE` 从句过滤行。
- 受影响的行数大于 *N*。
- 修改有一定数据量的表。
- 对元数据表进行的更改由于太忙而无法实时更改。
- 具有默认值的新列。
- 创建 / 更改语句涉及某些数据类型，例如 BLOB（binary large object，二进制大对象）文件。
- 无索引的外键。
- 在特别敏感的表上进行操作。

为团队在生产环境准备的标记和防护措施越多，就越能增强团队的信心，从而加快开发速度。假设软件工程师签入了 456 号变更，该变更被视为具有重大影响而被标记。如果之前实施过此类变更并记录了相关文档，就可以对该操作应用这种变更模式，否则应该和数据库可靠性工程师一起建立一种新的变更模式。

1. 模式：锁操作

在大多数数据库中，新增列是常见的操作。根据所使用的 DBMS，这些操作可以快速而简单，不需要锁定表。有些 DBMS 则需要重建整个表。当新增列时，可能需要设置默认值，这无疑会对数据库产生重大影响，因为该默认值必须写入已有的每一行数据中，之后才能完成变更并释放锁。

避免某些锁操作的一个办法是利用代码，例如：

- 添加空列；
- 执行回归测试；
- 在访问时利用 `select` 语句中的条件确定一行是否需要更新，而不是作为批处理语句执行；
- 监控何时该属性已经全部更新完毕，可以删除条件代码。

对于某些操作，锁定对象是不可避免的。在这种情况下，必须为工程师提供一种自动的或手动的模式。该模式可能是通过触发器和表重命名进行在线变更的工具，也可能是利用代理和故障转移在离线节点上逐个应用变更的滚动变更。

有两种方式：一种轻量，另一种步骤更多。这样，只需对有重大影响的变更采用复杂模

式。然而，这可能导致过度依赖一个过程，另一个过程未被实践而问题重重。因此，最好采用对所有锁定操作最有效的流程。

2. 模式：占用资源较多的操作

根据所要执行的操作，有很多模式可以利用。

对于数据修改，当需要执行大量操作时，工程师可以采用一种简单模式：通过执行批处理操作进行限制。对于更大的环境，在用户登录时通过代码延迟更新或者查询，通常更有意义。

对于数据删除，软件工程师在其代码中可以使用**软删除**。软删除会将一行数据标记为“可删除”，这行数据在应用程序查询时会被过滤，并按需删除。这样就可以限制删除操作，异步删除这些数据了。对于大型数据集的批量更新，这种方式可能不可行。如果删除是按范围（比如日期或 ID 分组）定期执行的，则可以利用分区特性删除分区。删除一个表或者分区不会产生回退 I/O，从而减少资源消耗。

如果发现 DDL 操作（比如表更改）产生了足以影响响应时间的 I/O，则应将其视为危险信号，即容量可能已达到极限。理想情况下，要和运维人员一起向存储系统中添加更多资源，但如果这不可行或者被推迟，则应将这些 DDL 操作视为阻塞操作，并对其应用适当的模式。

3. 模式：滚动变更

如前所述，让工程师可以在集群中的节点上逐步应用变更是有意义的。这通常称为**滚动升级**，因为是在逐节点执行变更。根据集群能从任意节点写入，还是只能从某个节点写入，实现滚动升级的方式会有所不同。

在任意节点可写入的集群（比如 Galera）中，可以通过从服务目录或者代理配置中移除节点，让该节点不再提供服务。节点没有流量之后，便可执行变更操作。然后重新配置节点或将其注册到服务目录中，使其重新提供服务。

在只有主节点可写的集群中，只有一个节点提供写入服务，可以像在任意节点都可写入的集群中那样，让从节点停止服务。但是，当除主节点外的所有节点都更新后，需要进行故障转移，将写切换到已经更新的节点上。

显然，这两种选择都需要大量的编排工作。了解哪些操作耗时耗力、需要滚动升级，对于选择数据库非常重要，这也是许多人在研究如何让数据库 schema 变更影响变小的原因。

4. 变更测试

尽管这似乎是显而易见的，但必须认识到，如果变更集的实现细节修改了，那么在部署到后集成环境（包括生产环境）前，相应的变更修改必须签入并完成集成测试。

5. 回滚测试

除了测试变更及其影响，数据库可靠性工程师及所支持的团队还要考虑变更或部署失败时，回滚部分或全部变更集的策略。数据库变更脚本应该和变更同时签入。某些变更（比如建表

操作）可能会自动生成默认值，但是必须考虑已生成的数据，所以不建议通过简单地删除对象来恢复。在已有数据写入且必须恢复时，重命名表可以让这些数据仍然可访问。

变更模式可以简化定义回滚的过程。有效的回滚脚本是集成和部署的一个重要准入条件。要验证回滚脚本是否有效，可以利用下面的部署和测试模式：

- 应用变更集；
- 快速集成测试；
- 应用回滚变更集；
- 快速集成测试；
- 应用变更集；
- 快速集成测试；
- 长时间的周期性测试。

回退测试和恢复策略测试同样重要，必须将其融入所有构建过程和部署过程。

8.4.4 手动或自动化

应用变更模式的另一个优点是，变更模式可以实现自动批准和部署，无须等待数据库可靠性工程师团队审查和执行。此外，有些模式可以自动标记为需要数据库可靠性工程师处理，以便在自动化流程之外实施。

不必刻意追求自动化，尤其在处理重要数据时。虽然根据社区最佳实践，任何频繁执行的操作都应该尽可能自动化，但失败的自动化产生的影响会抵消其带来的好处。如果已经构建了一个环境，并且该环境经过测试，还具有可靠的回退机制、快速的恢复流程以及成熟的工程师，那么可以采用变更模式，并将变更相关的应用自动化。只要朝着标准化模型、一键部署和回退，以及建立防护措施的方向前进，便是在向目标迈进。

8.5 小结

本章讨论了数据库可靠性工程师团队在软件开发的各个阶段（开发、集成、测试和部署）为软件开发团队创造价值的方法。再怎么强调也不为过，这在很大程度上依赖数据库可靠性工程师、运维人员以及软件工程师团队的通力合作和良好关系。在此过程中，数据库可靠性工程师必须扮演老师、外交家、协调者和学生的角色。在培训人员和人际关系中投入得越多，应用本章所介绍的知识时收获就越多。

安全性是发布管理中不可或缺的一部分。数据是基础设施攻击中的重要目标之一。任何变更和功能都可能包含漏洞，必须将其考虑在内并设法减少。第 9 章将讨论数据库可靠性工程师如何在安全规划和流程中做出贡献。

第 9 章
安全

保障数据安全是数据库管理员的一项重要工作。数据是公司最重要的资产，数据安全与数据恢复一样至关重要。然而，安全事故和攻击事件日渐频繁，只要打开新闻，就会看到成百上千（甚至上百万）的用户档案、信用卡、电子邮件等信息被窃取并转卖的安全事件。

在分工明确的情况下，数据库管理员只负责安全控制，而强化隔离措施以及识别安全问题则由他人负责。但作为公司数据的管理员，数据库可靠性工程师要全面考虑安全这项工作。

前面讨论了持续部署流程、云计算环境以及基础设施即代码，攻击者为了获取想要的数据，可能会将这些作为攻击对象。本章为数据库可靠性工程师起草了一份数据库安全规范，以适应当今组织和基础设施的需要；然后将依据该规范讨论潜在的攻击对象、减轻攻击影响的方法和策略，以及数据库可靠性工程师可以拥护的整体模式。

9.1 安全的目标

如第 7 章所讨论的，数据安全与数据恢复同等重要。根据数据敏感性的不同，数据被窃与数据损坏可能一样糟糕。正如数据恢复的应用范围较广（不仅用于紧急恢复），安全功能也有很多用途。

9.1.1 防止数据被窃

这是数据安全的典型应用场景。大部分情况下，对于每个存储数据的机会，都可能有人试图非法访问数据。有些人（内部的或外部的）想获得数据库访问权限以便倒卖用户数据，或窃取商业机密，或单纯想利用获得的数据搞破坏。可能遭到这类攻击的数据包括：

- 线上数据库中的数据；
- 数据存储之间移动的数据；
- 备份和归档的数据；
- 从数据存储发送到应用程序或者客户端的数据；
- 内存或应用程序服务器上的数据；
- 从应用程序经网络发送给用户的数据。

如前所述，盗窃者并非都来自外部。了解内部系统并获得访问授权的内部用户，甚至比很多来自外部网络的攻击者更具威胁。无论攻击者来自哪里，数据库可靠性工程师都需要和安全管理员、运维人员及软件工程师一起，保证数据被正常读取、复制和传输。

9.1.2 防止故意破坏

恶意破坏有时单纯是为了打击公司。攻击者会用各种方式破坏公司的数据库和数据，包括损坏或操纵数据、关闭数据库或消耗各种 IT 资源直到不可访问为止等。他们会采用 DoS（denial of service，拒绝服务）攻击，利用漏洞关闭数据库，或者获取并操纵数据。对这类攻击而言，虽然可以利用备份进行恢复，但毁坏备份比毁坏线上存储更容易。

9.1.3 防止意外损坏

尽管安全通常指防止坏人入侵，但避免有人无意进入错误的环境，或使用错误的 schema 和对象，从而导致数据意外损坏同样重要。防护措施可以阻挡外来者，也可以警告人们他们正在进入禁区。对于防止数据被窃而言，内部恶意破坏者更具威胁，他们利用工具和证书可迅速制造灾难。

9.1.4 防止数据泄露

即使没有有意或无意的入侵者，也依然存在风险。在复杂的分布式和解耦的系统中，很容易因 bug 或证书配置错误，导致在用户浏览器或者电子邮件中明文公开日志中的敏感数据，甚至允许未经授权的人以用户身份登录他人账户。这种信息泄露会导致用户质疑公司托管和保护数据的能力。

9.1.5 合规与审计标准

组织受到众多用于保护用户和个人利益的标准和法律法规的严格监管。对组织进行相关标准的培训并确保组织行为符合规定，是安全范畴的工作。这是一项吃力不讨好的工作，并常常使专注于新功能和组织扩张的人感到挫败。不过，如果组织不希望因此被罚款甚至倒闭，就需要重视这项工作。

9.2 数据库安全即功能

通过跨职能部门的人际关系和方法来保障数据库可靠性，是本书一直倡导的理念。数据库可靠性工程师已逐渐成为公司其他团队的联络人、主题专家和培训师。随着开发团队规模的快速增长，这是唯一的发展途径。公司内的信息安全专家通常人手不足，这使得数据库可靠性工程师和信息安全团队在面对不断变化的状况时，难以有效地保护公司数据。

如第 8 章所述，本书致力于通过自助服务、培训和择优选择的方法，来实现安全、有效、快速的开发。开发人员处在保障数据安全的第一线，他们将以类似的方式实践本书所讨论的方法。安全性必须最先集成到应用程序和基础设施的开发流程中，而不是在发布时作为合规性条款的旁注或可选项。

怎样才能做到这一点呢？可以使用本书介绍的工具来完成此任务。

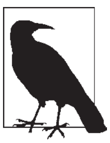
9.2.1 培训与合作

第 8 章详细讨论过这一点，可总结为如下 3 个方法：

- 鼓励沟通交流；
- 建立专业领域知识库；
- 通过结对和评审方式开展合作。

这样做是为了教开发人员如何更有效、更安全地开发自己的防御系统，以便抵御针对公司数据存储的攻击。这样可以提高应用程序的性能和效率，减少由于实现不佳和设计糟糕而导致的停机和服务质量下降，并提高开发团队的开发速度。同样，也要持续开展数据库安全培训，包括如下几点。

- 安全的数据库访问配置和控制。
- 有效利用安全功能，比如加密、细粒度访问控制和数据管理。
- 数据库对监控工具、日志、遥测设备等公开了什么数据，以便发现恶意和破坏性行为。
- 数据库特有的漏洞必须在其他层进行管理，包括发布 CVE 新条目后的升级。



CVE

CVE 代表公共漏洞和暴露（common vulnerabilities and exposures，参见 CVE 数据库）。你也可以关注感兴趣的主题（比如 SQL 注入漏洞）。这些资源非常好，有助于你了解新发现的漏洞和已知漏洞。

通过持续培训和协作，数据库安全在公司内逐渐成为一个经常讨论的话题，不断探索和深入研究。

9.2.2 自助服务

为数据库安全创建自助服务，是构建严谨、成熟的安全流程的下一步，该流程可以随着开发团队的规模和功能研发速度而扩展。数据库可靠性工程师不可能亲自审阅每一个新功能、新服务以及新数据库；相反，随着待办事项越来越多，他们会发现自己经常成为审阅请求的瓶颈。所以，请和信息安全团队一起建立可复用、经过验证的安全模式，这样工程师可以在需要的时候签出该模式，从而实现可扩展的安全流程。

如第 5 章所述，基础设施即代码可以实现所有数据存储的部署模板（经过验证的），这些模板可以应用于任何即将投入使用的数据存储中。这意味着，数据库可靠性工程师的大部分时间将花在建立这类黄金标准、进行漏洞研究，以及修改和更新平台的部署模板上，以减轻漏洞的影响，具体包括以下内容。

- 批准软件构建号。
- 删除数据库的默认账号和密码。
- 关闭不必要的端口。
- 建立有效的访问受限列表，以减少数据库的入口点。
- 删除允许通过文件系统或网络实施攻击的功能和配置。
- 安装并设置 SSL（secure sockets layer，安全套接字层）的通信密钥。
- 用于检查和实施密码策略的脚本。
- 配置审计和日志转发，以确保所有访问都可以被审查，并且不被篡改。

通过签入以上内容，并保证其在部署新数据存储时可用，你可以提前批准采用了这些黄金标准的基础设施。这样数据库可靠性工程师和信息安全团队就不用再花时间检查和报告已经安装补丁的漏洞了。

除基础设施自助服务外，日志、验证、密码散列、加密等代码库也可以签入并启用，也包括客户端软件。

构建自己的数据库客户端

与基础设施自助服务类似，提供客户端和库的自助服务是有效的技术。厂商提供的客户端通常暗含了一些变通方案，而这些方案日后可能暴露未知漏洞。这些客户端为了向后兼容经常使用旧版本的协议。而编写自己的客户端可以降低未知因素带来的风险，并且能控制数据库的核心层：访问层。

9.2.3 集成和测试

集成和测试往往能尽早发现漏洞。尽量不要在开发流程的后期才发现漏洞，因为此时修复漏洞的成本会成倍增加。另外，鉴于占有测试和集成服务器的潜入者可以轻易地绕过所有测试并注入恶意代码，集成和测试过程也是漏洞被利用的高危风险点。

集成过程中，获批的标准测试可以自动检验有无引入漏洞。标准测试包括（但不限于）下面内容：

- 数据库访问功能中的 SQL 注入漏洞；
- 测试验证层常见缺陷，包括明文通信、纯文本证书存储和以高级管理员身份建立连接；
- 测试新代码的漏洞，比如缓冲区溢出。

除了签入后立即触发的测试，还可以定期执行更细致的异步测试。这些测试包括应用级别的渗透测试，以及在网络中对漏洞进行的严格测试。这些漏洞可以由未经身份验证和已验证的方式被利用，从而获得数据库或操作系统的访问权限。

9.2.4 运维可见性

将安全功能的输出集成到标准日志和监控系统及其输出中至关重要。这些数据来自整个系统的各处，包括应用程序层、数据库层和操作系统层，详细内容见第 4 章。

1. 应用程序层监控

追踪发送到数据库的所有成功和失败的 SQL 语句，对于发现 SQL 注入攻击非常重要。SQL 语法错误是重要迹象，预示着有人或某个工具正在尝试通过应用程序将计划外的 SQL 语句传给数据库。在正常运行、经过测试的应用程序中，语法错误是很少见的。类似地，仔细研究就会发现，SQL 注入模式经常会出现一些字符，此类字符往往预示着攻击正在进行，包括 `UNION` 和 `LOAD_FILE` 语句。9.3 节将进一步讨论。

审计数据应该围绕 PII 和关键数据进行收集。利用元数据将 API 端点标记为是否属于 PII 或影响是否严重，以便收集关于访问、修改、删除数据（比如密码、电子邮件、信用卡或文档块）等细粒度的信息。尽管数据库层也有审计功能，但应用程序层的审计便于让合适的人员获得访问权限，以判断应用程序代码是否被无意或故意滥用。

2. 数据库层监控

在数据库层应该记录所有活动，并推送到监控系统以便分析。下面是一些值得关注的活动。

- ❑ **配置变更**

  在文件中或内存中会发生这类变化，配置文件变更可能会为攻击者打开“方便之门”。

- ❑ **数据库用户变更**

  对于所有权限或密码变更以及新增用户，都需要检查签入代码和集成的相应迁移，否则这些变更可能会导致安全漏洞。

- ❑ **所有的增删改查**

  数据库级别的审计日志为应用程序审计日志提供了良好的补充。大量的查询或修改、不符合预期的用户查询和大结果集合都是潜在问题的征兆。

- ❑ **新数据库对象，尤其是存储的代码**

  新增或修改的函数、步骤、触发器、视图以及 UDF（user-defined function，用户定义的函数），应该和数据库迁移相关联，因为它们可能是漏洞被利用的征兆。

- ❑ **成功或失败的登录**

  任何数据库都有预期的流量模式。应用程序的用户通常来自特定的主机组，一般不应该直接登录数据库。在某些环境中，如果登录不是来自应用程序服务器、代理或其他已授权的客户端，则可以将对该数据存储的访问标记为可疑行为。

- ❑ **补丁与二进制文件变更**

  通过网络缓冲区溢出或利用其他漏洞，以操作系统授权用户身份安装热补丁，这些变更可能会预留“后门”或注入潜在的恶意代码。

3. 操作系统监控

和数据库一样，也要仔细监控和记录操作系统的活动，包括以下内容。

- ❑ **配置变更**

  类似于数据库变更。

- ❑ **新增的软件、脚本和文件**

  在临时目录、日志目录或其他预期放置新文件的目录之外的区域，出现新的或修改过的软件、脚本或文件，往往是不好的征兆。定期将这些文件和黄金镜像做对比，可以发现恶意行为。

- ❑ **成功和失败的登录**

  类似于数据库登录。

- ❑ **补丁与二进制文件变更**

  与数据库的此类变更相同。

全面的数据收集结合有效的对比和异常检测工具，对于发现利用已知安全问题的恶意行为至关重要。能将所有攻击者挡在门外，并且及时更新的全面安全策略并不存在，所以有效的监控系统是必需的。数据库可靠性工程师团队必须和运维团队、信息安全团队以及软件工程师全力协作确保系统安全。

9.3　漏洞和漏洞利用

前面从高层次讨论了数据库可靠性工程师在安全方面的主要工作职责，即帮助构建可扩展的安全功能。下面讨论各种潜在威胁，数据库可靠性工程师在公司中进行安全培训，构建自助服务平台，建立相应的监控、模板和脚本时，必须考虑这些潜在的漏洞并做好准备。

在对威胁建模时，分类和划分优先级很重要，结合其他因素进行权衡并确定优先级更重要，在这方面已有一些结构化的方法。微软已经公开了其威胁分类模型 STDRE 和威胁评级模型 DREAD。

9.3.1 STRIDE

STRIDE 是一个分类方案，根据漏洞类型（或攻击者的动机）来描述已知威胁。STRIDE 这个名称取自下面每个分类的首字母。

- **假冒身份**（spoofing identity）
  假冒身份即冒充其他用户进而绕过访问控制。因为大部分多用户的应用程序最终以某个用户的身份访问数据库，所以假冒身份有巨大的风险。
- **篡改数据**（tampering with data）
  除了假冒身份进行操作，用户还可以通过应用程序的 POST 行为修改数据。数据验证和用于管理的 API 对于防止这种情况发生非常重要。
- **抵赖**（repudiation）
  如果没有合理的审计，客户和内部用户可能会对他们所采取的行动提出异议，这会导致纠纷、审计失败以及无法发现恶意行为，进而造成财务损失。
- **信息泄露**（information disclosure）
  用户的隐私信息可能被公开，落入竞争对手或恶意买家之手。这也包括意外的信息泄露。
- **DoS**（denial of service）
  应用程序和某些基础设施组件也可能成为 DoS 攻击的目标，一些非常消耗资源的操作和来自全球的大量请求都可能导致 DoS 攻击。
- **权限提升**（elevation of privilege）
  用户可能获得更高的权限，甚至以 root 身份访问服务器。

9.3.2 DREAD

DREAD 威胁评级模型可以根据威胁的风险分析风险和划分优先级。DREAD 算法用于计算风险值，它是下面 5 类风险的平均值。

- **潜在损失**（damage potential）
  如果漏洞被利用，会造成多大损失？
  - 0：没有损失。
  - 5：用户数据被盗或受影响。
  - 10：整体系统或数据被摧毁。

- ❑ **可重现性（reproducibility）**

  重复攻击的难度有多大？

  - 0：即使是应用程序的管理员也很难重现或不可能重现。
  - 5：需要一到两个步骤，也许需要成为授权用户。
  - 10：只要有浏览器和地址栏就够了，不需要任何验证。

- ❑ **可利用性（exploitability）**

  利用该威胁的必要条件有哪些？

  - 0：高级编程和网络知识，定制或先进的攻击工具。
  - 5 ：网络上已有的恶意软件，或者通过攻击工具可以轻松利用漏洞。
  - 10：仅需一个 Web 浏览器。

- ❑ **影响的用户（affected users）**

  影响了多少用户？

  - 0：一个都没有。
  - 5：有一些，但非全部。
  - 10：所有用户。

- ❑ **可发现性（discoverability）**

  发现这个漏洞的难度有多大？

  - 0：几乎不可能，需要源代码或者管理员权限。
  - 5：可以通过猜测或通过监控网络轨迹发现。
  - 9：这类漏洞的细节已在网上公开，通过搜索引擎很容易搜到。
  - 10：相关信息在浏览器地址栏或者表单中可见。

当逐个考虑每个潜在的攻击媒介时，通过这种分类，可以决定将精力和资源集中在哪里。

9.3.3 基本预防措施

下面讨论各种可行的预防措施，并提供各种数据存储中的示例，其中包括之后将讲解的通用缓解技术，其中一些技术适用于多个类别。

- ❑ **配置**

  从数据库中删除所有不必要的功能和配置。许多数据库系统功能丰富，而大多数应用程序只能用到其中一小部分功能，关闭不必要的功能可以减少攻击媒介。

- ❑ **打补丁**

  持续扫描数据库的漏洞并打补丁。保持最新状态将降低漏洞被利用的风险。

❑ **删除不必要的用户**

默认用户和密码是众所周知的，因而会带来很大的风险。

❑ **网络和主机访问**

使用防火墙和安全组，能最大限度地减少可以通过端口访问数据库的主机组。同样，通过身份和权限限制，最大限度地控制对系统的访问也是当务之急。

> **使用默认值的风险**
>
> 在撰写本文时，监听公共 IP 的 MongoDB 和 Elasticsearch 数据库的漏洞造成了广泛的影响，而此类问题是可以避免的。2015 年，Shodan 上的一篇文章“It's the Data, Stupid!”揭露了 3 万多个 MongoDB 实例可公开访问，因为默认监听 IP 为 0.0.0.0 并且未开启身份验证。仅仅因为没人关注这些软件早期版本的默认值，就导致超过 595.2 TB 的数据被公开。

下面讨论以下内容：

- DoS 攻击；
- SQL 注入；
- 网络和身份验证协议。

9.3.4 DoS攻击

DoS 攻击通过向服务或应用程序发送大量请求，使其资源饱和，无法处理实际用户的请求，从而使服务或应用程序不可用。这类攻击的形式通常是，通过客户端的分布式网络发送海量请求，以耗尽网络带宽。另一种形式是耗尽特定服务器或集群（例如数据库）的资源。通过耗尽 CPU、内存或磁盘资源，使得关键服务器变得无响应，从而导致依赖该服务器的所有服务都不可用。

这些攻击通常不是毁灭性的，因为不会损坏或窃取数据，其目的仅仅是让服务宕机，以便造成干扰、打击竞争对手，或是在其他攻击发生时分散信息安全团队和运维团队的注意力。

大型网络泛洪攻击是很常见的，因此大多数防御技术都将重点放在这上面。攻击者会选择攻击脆弱组件，这些组件可用资源虽少，但起着关键作用。在这种情况下数据库是“理想”的攻击目标，因此就出现了 DB-DoS（数据库拒绝服务攻击）。只需稍做手脚，就可以在数据库中执行大量逻辑，使资源饱和，这种饱和与正常的流量增长看起来差别不大。

DB-DoS 攻击的影响包括：

- 消耗用户连接，直到耗尽应用程序服务器的所有可用连接；
- 大量的差异化查询会干扰优化器，在查询优化期间，这些查询需要解析、散列和检查；

- 自动扩展资源，直到触发控制预算的措施，进而导致服务关闭；
- 从缓存中删除有效数据，导致大量的磁盘 I/O；
- 增加内存使用量可能导致页置换；
- 表空间和日志的增长用尽磁盘空间；
- 由于大量写入而导致复制延迟过多；
- 操作系统资源饥饿，包括文件描述符、进程或共享内存。

有些方法可轻松产生上述影响，最简单的方法就是利用应用程序本身的功能，例如：

- 往购物车中添加大量商品；
- 无输入或输入条件宽泛的搜索；
- API 响应较慢，表示查询可能未优化或未被索引，因此成为反复调用攻击的目标；
- 在输入表单中增加 `UNION` 语句,会导致大量的表连接和扫描(这也是一种 SQL 注入技术)；
- 对大型结果集进行排序；
- 创建边缘情况，例如论坛应用程序中大量的帖子，或社交应用程序中大量的好友。

与滥用应用程序功能类似，经验丰富的攻击者可以识别所用数据库，并找到关闭数据库的方法。例如通过错误登录来锁定用户，通过 SQL 注入来执行管理命令（可能会清除缓存），或发送格式错误的 XML 致使解析器溢出。

1. 缓解

除了讲过的标准缓解技术，DB-DoS 最有效的缓解方法与解决流量增长问题所采取的技术非常相似。下面会给出一个“铁三角”，它可以应对突如其来的资源利用率激增（无论来自合法流量还是非法流量）。这里没有提到自动扩容，因为容量始终存在上限，无论是由于硬件、软件、预算还是任何 DB-DoS 导致的。

2. 资源管理和负载削减

随着时间的推移，技术团队将了解应用程序的工作负载特征。可以合理地假设他们应该能够构建一套工具，来有效地应对负载激增。数据库可靠性工程师的职责是帮助培训和支持软件工程师了解这些工作负载，并确定工作的优先级，以便有效降低风险。工具如下。

- ❑ **客户端限流**

  通过限制请求和重试的间隔，可以防止机器人程序反复发送请求，并且可以有效减缓或阻止流量激增。可以利用普通计数器、指数下降和重试比例，或结合应用程序返回的有关超出服务等级配额的数据实现限流。

- ❑ **服务质量**

  还可以根据重要程度对应用程序的流量进行分类。将搜索这种开销大的查询请求标记为不太重要（因为这些请求可能会被利用），可以使应用程序强制执行服务质量配额，从而使 DB-DoS 更难以实现。

- **减少结果集**

  对于开销大的查询和 RPC（remote procedure call，远程过程调用），可能需要开发两条执行路径。对于正常负载，完整的执行路径没有问题；但是在高负载（例如在 DB-Dos 期间生成的负载）期间，可能需要减少扫描的行数或查询的分片数。

- **终结查询和笨办法**

  如果无法通过代码彻底解决这个问题，可能需要使用蛮力。结束长时间运行的查询，或调优数据库层性能以减少查询的资源消耗是非常有效的，但代价是缺乏控制力且用户体验可能很差，而这难以通过代码改善。

3. 持续改善数据库访问和工作负载

如果发生了 DB-DoS 攻击，可能需要数据库可靠性工程师和软件工程师手动清理数据库中开销大的查询。由于这些查询通常可能不是正常工作负载模式的一部分，因此根据数据库层资源的总消耗进行调优的方法可能会忽略这些查询。如果这些异常负载很少出现，则很容易被忽略。但是，狡猾的攻击者可能会发现这种漏洞并加以利用。这意味着，良好的性能优化方案应该找出开销最大的查询，并将其放入待优化队列，而无须考虑执行频率。

4. 日志和监控

无论上述措施效果如何，执着的攻击者仍有可能攻击你的数据库。有效的端点调用监控应该能够识别出尖峰，并通知分流团队限流甚至关闭。同样，如果存在没有上限的查询或活动，监控 `IN` 子句中的项目数、内存结构、永久表或临时表以及类似结构，有助于识别潜在问题。

重要的是要记住，除了窃取数据或故意破坏，还可能存在其他攻击形式。在规划安全功能时容易忽略 DB-Dos 攻击。接下来考虑另一个威胁。

9.3.5 SQL注入

SQL 注入是通过应用程序输入注入数据库代码（通常是 SQL）的一类漏洞利用方式，旨在绕过安全防护，在数据库中执行非应用程序预期的代码。SQL 注入可以利用缓冲区溢出漏洞关闭数据库、执行 DB-DoS，或者在数据库甚至操作系统级别提升用户权限。

SQL 注入的另一个攻击媒介是数据库中存储的代码，此类代码（例如存储过程）通常可以在提升权限的情况下执行任意语句。内部用户或设法通过凭证猜测或嗅探获得访问权限的人，都可以利用此漏洞。

通过 `UNION` 语句，还可以利用 SQL 注入来访问数据，从而从其他表中获取与查询表具有相同列数的数据集。例如搜索表单查询具有 5 列的表，则在表单中注入 `UNION` 语句可以将结果集添加到任何具有 5 列的表中，这样无须利用漏洞便能轻易窃取数据。

1. 缓解

要在应用程序层减轻 SQL 注入的影响，首先要对软件工程团队进行培训。编码时，软件工程师必须避免动态查询，并且必须防止恶意 SQL 输入修改查询语句。

2. 预编译语句

第一步是确保工程师使用预编译语句。预编译语句也称**参数化语句**。在预编译语句中，查询的结构是预先定义的。然后将表单输入与查询语句中的变量绑定，与之相反的是在运行时动态定义和构建 SQL。预编译语句比较安全，因为恶意者无法修改查询逻辑。如果发生 SQL 注入，则将注入的 SQL 用作比较、排序或过滤的字符串，而不是将其视为单独的 SQL 语句。

示例 9-1　Java 中使用预编译语句的示例

```
String hostname = request.getParameter("hostName");

String query = "SELECT ip, os FROM servers WHERE host_name = ? ";

PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, hostName);
ResultSet results = pstmt.executeQuery( );
```

3. 输入验证

有时预编译语句起不到保护作用。动态表名称或输入是不能预编译的，这时就需要针对要保护的目标来验证输入。在这种情况下，应用程序通过定义好的列表，检查表名、`Desc` 和 `Asc` 关键字的有效性，以便确定查询能否安全地执行。还可以验证（美国）邮政编码为 5 个整数、字符串没有空格，并且长度在特定范围内。

4. 减少损害

在预防措施无法保护应用程序输入的情况下，还可以在应用程序外部采取其他缓解措施，以减轻 SQL 注入的影响。由于无法保证不会发生 SQL 注入，因此最好建立深度防御机制，这包括给数据库二进制文件打补丁，以减少可能被利用的漏洞；消除不必要的存储代码以及（应用程序使用的）数据库用户的不必要特权也很重要。此外，为每个应用程序分别指定数据库用户，可以减轻账号遭劫持所造成的损害。

5. 监控

类似于其他故障或功能异常，测量和监控数据注入对于降低风险至关重要。确保记录和分析转储文件、栈追踪以及查询日志模式（匹配联合查询、分号和其他指示 SQL 注入的字符串）也至关重要。

SQL 注入很容易预防，但是它是利用数据库漏洞的常见方法之一。对于不断壮大的大型软件工程团队而言，持续培训、协作的平台以及经信息安全团队和数据库可靠性工程师批准的共享库，是保证安全性和开发速度所必需的。

9.3.6 网络和身份验证协议

恶意者有多种方法利用各种通信协议攻击数据库服务器。如果网络协议中存在漏洞，利用该漏洞可能会直接获得服务器的访问权限。曾有人通过“hello”漏洞（CAN-2002-1123）实施过攻击，该攻击利用了会话设置代码中的漏洞。在通过网络验证身份之后，也可能利用漏洞来获得操作系统或数据库访问权限或提升权限。同样，数据库协议可能遍布漏洞。实际上，一些服务器允许进行未加密的通信，这可能导致凭证被盗用。有时，漏洞可能导致在没有凭证的情况下发送身份验证信号。

9.3 节开头讨论的缓解技术，有助于减少数据库基础设施中的攻击媒介。使用这些技术，并让工程组织深入了解身份验证协议和数据库功能，可以确保配置尽可能安全。

潜在的攻击媒介及其缓解策略已概述完毕，下面讨论在发生入侵、权限提升之后，甚至数据库和服务器的完全访问权限已被获取的情况下如何保护数据。这就引出了加密。

9.4 数据加密

在某些情况下，不可避免地会有恶意者、悄无声息的侵入者甚至是无心的闯入者，可以访问本不该访问的数据。即便已经锁定了所有已知网络和操作系统路径，尽量限制了每个用户的访问权限，修复了所有已知的漏洞，并封堵了所有可能被利用的应用程序漏洞，仍需要为不可避免的情况做准备。

加密是使用一组商定的密钥转换数据的过程。从理论上讲，只有拥有密钥的人才能将加密数据解密为可用格式。加密通常是数据防御的最后一种形式，即使数据已被盗，落入恶意者手中。

下面从 3 个层次讨论加密：

- 传输中的数据；
- 数据库系统中的静态数据；
- 文件系统中的静态数据。

这些都是潜在的攻击媒介，可以通过加密来保护。但是，这些加密方法增加了开销和成本，大多数组织需要谨慎考虑和权衡。

在每一层中，必须考虑数据的另一个维度——数据类型。数据有是否敏感之分，稍后将按照敏感性将数据分类。如果要存储和管理此类数据，数据库可靠性工程师团队应与信息安全团队和软件工程师团队紧密合作，以确保各方都了解义务和标准，并且构建的自助服务平台、库和监控符合这些规定。数据有不同的形式，下面介绍一些常见的、可管理的合规数据形式。

9.4.1　财务数据

这包括账号和相关的能用于身份验证和识别的数据，交易历史数据，以及可以揭示个人或组织的财务状况的数据，例如信用评分、财务报告和余额等。在美国，财务数据受到众多法律、机构和标准的监管，包括关于信用卡数据的 PCI DSS、GLBA、SOX/J-SOX、NCUA、数据隐私和数据驻留法以及《爱国者法案》。其他国家也有类似的监管主体。

9.4.2　个人健康数据

有关患者及其健康状况的信息属于此类别。它包括个人身份信息，例如社保号、姓名和联系信息，以及有关患者健康状况、治疗过程的数据以及保险信息。美国的健康数据主要受 1996 年的 HIPAA 法案监管。

9.4.3　个人隐私数据

通常称为个人身份信息。它包括社保号、地址、电话号码和电子邮件。此类信息泄露可能导致身份盗用和骚扰。1974 年的《隐私法》是美国目前针对这些数据制定的标准的基础。

9.4.4　军事数据或政府数据

任何与政府运作或人员有关的数据，都被视作非常敏感的，有关军事行动和人员的信息更是如此。对于任何支持和存储此类数据的组织，都有非常严格的监管流程。

9.4.5　机密或敏感的业务数据

这包括必须保密以保护企业竞争力的任何数据，例如知识产权、商业机密、财务和绩效 / 活动报告、用户信息和销售信息。

了解各个数据存储中数据的性质，对于为加密和保护做出适当的决策至关重要，这也是组织协作和培训之所以重要的原因。否则，在快速发展的组织中，不知情的工程师很容易将敏感数据存入尚未被保护的数据存储中。

应该遵守一些基本准则。虽是老生常谈，但前述 MongoDB 采用默认值的案例体现了提醒的重要性。

- 直接与数据库交互的 Web 管理界面，应始终使用 SSL 或安全代理服务。本书使用 SSL 指代两种协议。TLS（安全传输层协议）是 SSL 3.0 的后续版本，大多数人将两者都称为 SSL。TLS 1.0 存在安全漏洞，不够安全 [1]。
- 应该使用 SSH2 或 RDP（remote desk protocol，远程桌面协议）连接服务器。

注 1：参见“Differences between SSL and TLS Protocol Versions”。

- 与数据库的管理连接应使用单独的管理网络，并且如果数据库支持，应使用 TLS 1.1 或 TLS 1.2。
- 所有 SSL 协议都应使用适当强度的加密密钥。会话的加密强度取决于服务器和客户端之间协商的密钥。

首先介绍传输中的数据的加密。

9.4.6 传输中的数据

数据肯定会在网络中传输。对于希望保护数据的数据库可靠性工程师而言，这是不可避免的事实。就像金钱或贵重物品必须用装甲车运输，数据也需要安全传输。传输是非常脆弱的环节，根据传输过程中的位置，这种脆弱性会放大。

在深入研究之前，数据库可靠性工程师必须了解加密套件的各种组件以及最佳实践。

1. 加密套件剖析

每个数据库服务器将通过协商好的**加密套件**与客户端通信。重要的是理解特定数据库加密套件的含义，以确保根据所需的安全级别保护数据。信息安全团队应该制定标准；如果这是你的责任，那么你需要理解特定数据库实现的含义。以下是加密套件的一个例子：

```
ECDHE-ECDSA-AES128-GCM-SHA256
```

该套件的第 1 部分 `ECDHE` 是密钥交换算法。这个例子采用了临时密钥进行密钥交换——**椭圆曲线版本**（elliptic curve version），还可以采用 RSA、DH 和 DHE。临时密钥交换基于 Diffie-Hellman[2]，并且在 TLS 初始握手期间，每个会话使用一个临时密钥。这种方式提供了 PFS（perfect forward secrecy，完美的前向保密），这意味着服务器长期签名密钥的妥协不会对过去会话的机密性造成损害。当需要使用临时密钥时，服务器会使用长期密钥（证书中的密钥）来签署临时密钥[3]。普遍认为 DHE 比 EDHE 的安全性更高，应该优先使用。

第 2 部分 `ECDDSA` 是签名算法，对密钥交换参数进行签名。这里，RSA 优于 DSA 和 DSS，签名熵的不同可能导致后者非常弱。

`AES128` 指的是套件中使用的加密算法，在本例中是具有 128 位密钥的 AES（advanced encryption standard，高级加密标准）。然后是加密的操作模式，在本例中是 GCM（Galois/Counter 模式），提供经过验证的加密。GCM 只支持 AES、Camellia 和 Aria，因此这些加密方式更理想。对于 AES，美国国家标准与技术研究所（NIST）提供了 3 种选择，每种选择的块大小为 128 位，但密钥长度不同，分别为 128 位、192 位和 256 位。

注 2：Whitfield Diffie, Martin E. Hellman. New Directions in Cryptography.

注 3：参见“Transport Layer Protection Cheat Sheet”。

最后，SHA256 是 MAC（message authentication code，消息鉴别码）的函数 e。SHA256 是验证机构用于签署证书以及 CRL（certificates revocation list，证书撤销列表）的散列 MAC（HMAC）函数。我们使用该算法创建主 secret，消息的接收者以此验证内容是否正确。一端发送消息，并从另一端接收和验证消息之后，就可以开始收发应用程序的数据了。SHA-2 是该算法的首选实现，它包括 4 种散列函数：SHA224、SHA256、SHA384 和 SHA512。

评估数据库的 SSL 实现时，理解加密算法的列表顺序非常重要，因为该列表决定了寻找客户端和服务器都可用的加密算法时扫描的顺序。

评估加密通信的需求时，不仅要考虑先前讨论的数据类型，还要考虑传输路径和边界，具体分为：

- 内网通信；
- 外网通信。

网络传输的每个方面都需要考虑并需要做出一系列假设，其中每个假设都有一组需求，每个需求都需要相应的实现。

2. 内网通信

安全子网中的通信通常假定网络层是安全的，因此无须进一步保护通信通道（网络连接本身）。这意味着在安全的网络环境中，应用程序服务器请求或发送数据、服务器之间的数据复制以及其他网络通信，不需要对通信进行加密。这样很好，因为数据库大多数时候属于这种情况，而且加密是 CPU 密集型操作。话虽如此，如果数据库存储敏感数据，仍需要以某种方式加密，理想的做法是把数据存入数据库时在应用程序层进行加密。稍后讨论静态数据时会进一步讨论这个问题。

如果数据库中的敏感数据由于遗留问题或其他限制而无法保证安全，应该考虑加密所有数据库连接。

3. 外网通信

在组织的两个网络，或组织的网络与互联网之间进行通信，必须使用采用 IPSec 或 SSL 的 VPN（virtual private network，虚拟专用网络），无论传输的数据是否敏感。

类似地，对于大多数通信而言，客户端应该通过 SSL/TLS 与负载均衡通信。这样做确实增加了客户端和负载均衡的 CPU 开销，但是客户端与服务器之间极少以明文方式传输数据。

了解了 SSL 的需求之后，下面看看实现这一目标可供选用的架构。

4. 建立安全的数据连接

现代数据库系统通常在一定程度上支持 SSL，但也有例外，比如 Redis。确定数据存储是

否满足需求时，验证这一点非常重要。不应该缓存未加密的敏感数据。

需要指出的是，相比传言，SSL 开销通常非常小。大部分计算开销产生在启动连接时，很少会超过 2% 的 CPU 开销或增加 5 ms 的延迟[4]。一些加密方式（比如 AES）在大多数现代 CPU 中有内置指令，与基于软件的加密方式相比极大地提高了速度。

有一些分层的方法可用于保护连接。

基本的连接加密。在最基本的层面上，首先配置数据库服务器，以确保所有连接的通信都是安全的。同时，需要创建由认证机构签发的证书，该证书用于签发服务器公钥和私钥，客户端也使用相同的证书来生成公钥和私钥。在客户端存储密钥并适当配置服务器的情况下，所有连接都被视作加密的。

最佳实践和常识表明，密钥不能存储在容易被劫持的地方。这意味着，可以使用动态配置将密钥直接加载到应用程序的内存中，而不是直接存储在客户端的文件系统中。不过，还有更好的办法。

安全存储的密钥。尽管通过 SSL 保护连接是至关重要的第一步，但仍然存在漏洞。毕竟，如果参与者能够获得对客户端主机的访问权限，就可以利用使用这些密钥创建的连接来查询数据。使用密钥管理基础设施安全服务，比如 Hashicorp 的 Vault、亚马逊的 KMS（key management service，密钥管理服务），或其他任何类似的解决方案，可以将密钥的存储和管理与访问数据者分离开来。

此外，用于访问数据库的其他信息（比如用户名、密码、IP 和端口），可以存储在类似的远程服务中，以确保文件系统中不存储任何凭证，以防止在应用程序上下文之外获取并使用密钥。

动态构建的数据库用户。在前两个阶段的基础上，自然会采用相同的安全密码服务（比如 Vault），动态创建数据存储的临时账户，这允许应用程序主机注册并请求创建临时账户。设置只读之类的角色，有助于自动应用各种权限，而且这些用户的生命期有限，从而确保了任何可能被劫持的访问只能持续有限时间。此外，可以将用户映射到特定的应用程序主机上，以便审计查询和访问，而这在一组服务器共享用户名的环境中是具有挑战性的。

兼用 SSL 和 VPN 技术，应该能够根据存储数据的需求对所有通信路径加密。此外，还可以利用密码管理服务，来减少针对文件系统上配置文件和密钥的攻击，防止任何具有操作系统特权的人读取和滥用。这就是保护传输中的数据。下面继续讨论静态数据，首先是数据库中的数据。

注 4：参见 Sascha Wenninger 的文章“Why You Shouldn’t be Afraid of SSL Performance”。

9.4.7 数据库中的数据

数据库中的数据也称“使用中的数据”，应用程序、分析人员和处理流程必须能够访问这些数据。这意味着，任何加密解决方案只能允许通过身份验证的人访问数据，同时防止任何恶意访问。如果用户设法通过身份验证进入数据库，通常能够读取对应权限下的数据。

潜在的攻击者可以分为 3 类：

- 入侵者；
- 内部人员；
- 管理员。

入侵者访问数据库或其服务器来窃取有价值的信息。内部人员属于受信任的团队，在数据库或操作系统中有相应权限，可能图谋获取超出其权限的信息。管理员是在数据库或操作系统中拥有管理级别特权的人，他们利用这些特权来获取有价值的信息。

本节开头列出的许多数据需要进一步加密，以确保只有拥有特定权限的用户才能读取。下面回顾有用的选项，以及对应的特性和潜在的缺陷。在此提醒，加密标准和最佳实践与 SSL 加密中一致。

1. 应用程序级别安全性

在对威胁建模期间，这种方式将特定的表或列标识为需要加密。应用程序使用加密库对数据进行加密，然后将数据提交到存储系统，之后就像提交其他任何字符串或二进制数据一样提交数据。获取数据的方式也与之类似，应用程序知道在使用数据前将其解密。这种加密和解密可以使用 Bouncy Castle 和 OpenSSL 等库完成。

在应用程序级别使用库进行加密，提供了数据库级别的可移植性；即使后端存储发生变化，仍能像以前一样进行加密和解密。这样还能控制加密库。信息安全团队可以在 VCS 中维护共享库，供任何人使用，而且不再需要对这些代码进行合规性审计，因为这些代码已在整个组织内获得批准。最后，此方法允许选择性加密，不加密其他列和表，以便索引、查询和报告。

这种方法的主要缺点是，它并不是应用于数据库中所有数据的一劳永逸的方案。因此，当对新数据建模并将其添加到应用程序时，开发人员必须考虑是否对新数据加密，然后才能着手实现。这种方法还要求数据流水线中其他所有客户端在读取这些数据时进行解密。

应用程序级加密以牺牲开发速度为代价换取了最大的灵活性。

2. 数据库加密插件

加密插件是安装在数据库中的加密工具。此方法独立于应用程序，编写相应代码的工作较少。根据插件的不同，列级别的选择性加密、访问控制和审计通常是其特性。

通常不建议使用同一个密钥对数据库进行加密，因为用户可能会利用漏洞获得更高的访问权限以读取更多数据。例如具有加密密钥访问权限的内部用户，可以获得更高级别的用户访问权限，并访问其安全组之外的数据。使用不同密钥对不同安全组的表进行加密，可以确保用户只能解密其安全组中的对象。这意味着，任何加密插件都应该具有选择性加密和访问控制功能，这样才是有效的。

与应用程序级加密不同，加密插件确实会导致数据库之间的可移植性问题。因此，如果你在初创公司或需求经常变化的环境中工作，这种解决方案可能过于僵化。

3. 透明的数据库加密

有一些安全设备会加密 / 解密所有数据库通信。这种方法相对简单，易于确保数据被加密，但对所有数据加密无疑会增加开销。话虽如此，通用的底层方法可使开销最小化。

4. 查询性能考量

尽管加密数据很简单，但数据的查询效率可能低下，并且会影响 schema 和查询的设计。列或表级别的数据加密不易支持范围查询或字符串搜索。因此，此类查询必须考虑如何对数据进行过滤和排序。

大多数加密功能没有保留顺序，因此不能使用 B-Tree 索引。B-Tree 索引是对加密数据创建索引的标准方式，通常使用未加密的字段以支持高效过滤。例如可以使用日期范围过滤来减少扫描加密值所需的数据集。为了更有效地支持加密数据的查询，可以在 schema 中存储加密字段的 HMAC，并且为散列函数提供 key。后续的数据查询将使用 HMAC，因而受保护的字段不会以明文形式出现在查询中。这样，就可以查询数据库中的加密数据，而无须在查询中携带明文敏感信息，还可以防止没有创建 HMAC[5] 所需数据的用户操作数据库中的数据。

使用散列字段上的索引可以获得同等性能，但会暴露有关索引值的频率和基数信息。类似地，攻击者可以通过其在索引中的位置来推断有关加密值的信息，甚至可以搜索散列的出现次数。甚至可以通过长时间的访问来研究和分析数值随时间的变化规律，以此获取数据相关信息。例如在插入数据之后，掌握相关知识的人可以通过事件和索引中的位置来推断潜在值。

因此，可以根据数据的值添加干扰技术，以减少攻击者通过散列、散列之间的关系以及新插入的值来推断关联或值的可能。这种手段包括在每个插入中添加伪数据，或分批插入以防止对原子插入进行增量观察。

尽管这是对性能和安全性考量的高层概述，但很重要的一点是，要提出这些问题和一些缓解措施，以便在计划阶段将数据库加密视为要事。

注 5：Erez Shmueli, Ronen Waisenberg, Yuval Elovici, et al. Designing Secure Indexes for Encrypted Databases.

数据库中存储的数据最终位于文件系统上。日志、数据转储和备份也都存于文件系统上，保护数据时必须考虑这些问题。因此下面讨论文件系统级别的静态数据加密。

9.4.8 文件系统中的数据

通过加密传输中的数据和使用中的数据，提供了严密的保护，但攻击者仍有可能直接从磁盘、磁带或其他介质访问数据。像其他应对技术一样，该问题也有多种解决方案。在考虑解决方案时，重要的是要考虑数据量、CPU、读写对延迟的影响以及访问数据的频率。

对于文件系统中的数据，攻击者可能采用直接攻击策略或间接攻击策略。在直接攻击中，攻击者直接从数据库软件外部访问数据库文件，通过网络从废弃的存储设备或从备份设施中获取数据。在间接攻击中，攻击者可以从数据库使用的文件中获取 schema 信息、日志数据和元数据。

除了标准的网络和访问控制策略，还需要对文件系统数据进行加密，以确保即使有人绕过这些步骤，也无法访问重要数据。考虑存储加密时，必须考虑多层，这些数据是存储在文件系统和设备上的。

1. 文件系统上的数据加密

当数据存放在文件系统中时，可以自动对其进行加密，这种方式与 9.4.7 节讨论的类似。这通常适用于保存在文件系统中的数据，例如备份或数据转储文件。通过对这一层加密，我们始终可以了解关键文件的加密状态。

此外，数据可以分为多块，存储在多个存储设备上。对于敏感数据备份和大型数据转储，这是很好的选择，因为可以防止访问一个存储设备的人获取完整数据集。这种方式还可以使读写操作并行化，从而更快地恢复；但这种方案会增加额外的开发和维护成本，因而需要整体考虑。

2. 文件系统加密

因为大多数数据存储系统会为元数据、日志和数据存储创建单独的文件，所以必须在文件系统层提供加密功能。可以在文件系统之上叠加加密文件系统，直接在文件系统中通过内置加密机制进行加密，或在文件系统之下的磁盘块进行加密。

将加密文件系统叠加到现有文件系统之上后，就可以使用任何文件系统。这是一个非常灵活的选项，因为可以将其用于特定目录，而不是对整个卷进行加密。Linux 的此类系统包括 eCryptfs 和 EncF，这些系统要求手动或通过 KMI（key management interface，密钥管理接口）提供密钥。许多文件系统（例如 ZFS 和 BTRFS）也具有加密选项，但需要检查它们是否暴露了未加密的元数据。

块级别加密系统在文件系统之下运行，一次加密一个磁盘块。在 Linux 中，有 Loop-AES、dm-crypt 和 Vera，这些工具都使用内核空间设备驱动程序，并且在文件系统层之下运行。如果想对写入卷的所有数据都进行加密，而不管数据存储在哪个目录中，这些工具很有用。

这些解决方案都会影响性能，必须做出权衡。将日志、元数据和其他类似文件存储在加密的文件系统上是有意义的。但是，如何从数据库访问数据文件呢？许多用户发现应用程序级或列级加密为数据库提供了必要的安全。由于性能原因，数据库文件本身保持未加密状态。将此解决方案与日志、元数据文件和其他系统级文件的加密相结合，可以创建有效的多层解决方案，并且无损性能。

3. 设备级加密

还可以使用内置加密功能的存储介质。鉴于存在许多已知的漏洞，并且会增加存储成本，其价值存疑。这里的要点是，这一层肯定会为数据安全性增加防御深度。

如前所述，数据加密在设计、实施和审核方面需要大量工作。前面讨论了加密传输中的数据、数据库中的数据以及文件系统上的静态数据。数据加密是安全的最后堡垒，可在访问控制、代码强化和常规修补都失效时提供保护。认识到这一点的重要性，也就认识到了安全性的每一层都是脆弱的，因而必须在每个层级考虑数据保护，以构建合理的保护强度。在加密过程中，无论处于哪个层级，都可以考虑以下清单。

- 是否根据敏感性对所有数据进行了分类？
- 是否有加密套件的标准，是否对其进行了审核，以确保合规性？
- 是否在追踪和关注有关漏洞的新报告？
- 软件工程师、软件可靠性工程师、运维人员和数据库可靠性工程师中的新员工是否了解加密库和加密标准？
- 密钥是否得到了有效管理，包括轮换、移除和测试？
- 是否对关键组件（包括日志、备份、关键表和数据库连接）定期进行了自动化渗透测试？

像任何自动测试和手动测试一样，重要的是要认识到无法测试所有情况。这就是为什么通过关注高风险和容易被利用的特性，可以在连续的测试和改进过程中关注重点、划分优先级和及时反馈。

9.5 小结

本章不仅深入探讨了数据库安全性的层级，而且详细介绍了如何成为组织中的安全官。同样，数据库可靠性工程师不能独自负责该任务。在快速迭代的动态环境中，数据库管理员的想法根本行不通，因而需要上升到数据库可靠性工程师的思维。数据库可靠性工程师应该与本章提到的每个团队积极合作，用自己丰富的知识来创建自助服务平台、共享库和团队流程。

一如既往，本章努力为运维本身，以及为依赖数据库可靠性工程师的其他技术团队的有效协作与支持建立坚实的基础。第 10 章将重点介绍各种数据库持久性选项，以及它们如何运用关键技术来提供弹性、扩展性和高性能的数据存储和检索。届时会不时地提到到目前为止所奠定的基础。

第 10 章

数据存储、索引和复制

为了深入讲解数据存储，本书大部分篇幅在讨论运维。不同的数据存储之间最重要的共同点是：存储数据。本章将解释单个节点如何存储数据、如何分割大型数据集，以及节点之间如何复制数据。

本书着眼于可靠性和运维，主要介绍存储和访问模式，以便选择正确的基础设施以及了解其性能特征，确保数据库可靠性工程师拥有足够的信息来帮助团队选择合适的存储系统。若想进行更细致的研究，建议阅读 Martin Kleppmann 的《数据密集型应用系统设计》。

10.1 数据的存储结构

数据库通常通过表和索引来存储数据。表是主要的存储机制，而索引是用于加快访问速度的数据子集。随着数据存储种类的增多，这种情况发生了显著变化。了解数据存储的读写原理，对于配置和优化存储子系统和数据库至关重要。

在了解数据库如何存储数据时，实际上不仅需要评估原始数据的存储方式，还需要评估其检索方式。在大型数据集中，以合理的延时访问特定数据子集，通常需要专门的存储结构（称为索引），以加快数据查找和检索。因此，在考虑将数据放入磁盘和索引中的存储 I/O 需求时，还需要考虑检索数据的 I/O 需求。

10.1.1 数据库行的存储

大部分数据适合采用传统的关系型数据库。下面首先介绍关系型数据库，然后讨论其他一些可选的流行数据库。在关系型数据库中，数据存储在**块**或**页**中，与磁盘上特定数量的字

节对应。不同的数据库使用不同的术语。本书使用块代表两者，块是存储记录的最细粒度。Oracle 数据库将数据存储在数据块中。固定大小的页称为块，就像磁盘上的块一样。块是读写数据的最小单元。这意味着，如果一行为 1 KB，块大小为 16 KB，一次读操作将读取 16KB 的数据。如果数据库的块小于文件系统的块，那么将浪费 I/O，如图 10-1 所示。

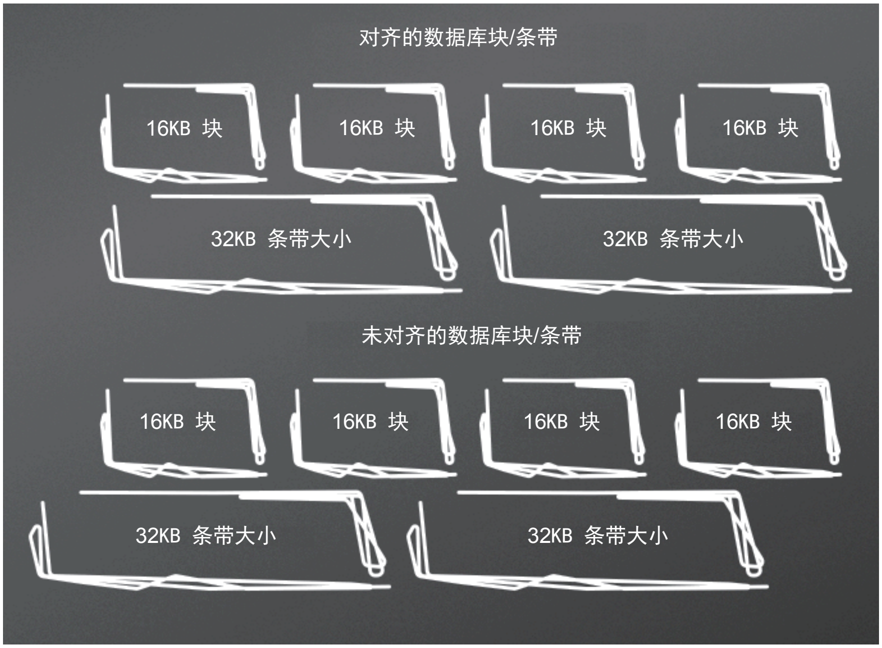


图 10-1：对齐或未对齐的块 / 条带配置

块还需要存储一些元数据，通常以头部和尾部或页脚的形式存储。这包括磁盘地址信息、该块所属对象的信息，以及块内行和活动的信息。在 Oracle 中，从 11g Release 2 版本开始，块的元数据总计为 84~107 字节；MySQL 5.7 版本的 InnoDB 中，元数据使用 46 字节。此外，每一行数据都需要自己的元数据，包括列信息、行分布的其他块的链接以及该行的唯一标识符[1]。

数据块通常组织成一个更大的容器，称为扩展区。出于效率原因，当表空间需要新块时，扩展区通常是分配单元。表空间通常是最大的数据结构，映射到一个或多个物理文件，它们可以根据需要在磁盘上进行布局。在直接映射到物理磁盘的系统上，可以将表空间文件分布在不同磁盘上以减少 I/O 竞争。在本书所关注的范例中，不一定需要这种 I/O 优化。条带化和镜像条带化的大型通用 RAID 结构可以最大限度利用 I/O 吞吐，而无须花费大量时间进行微调。然而，假设可以进行快速恢复和故障转移，将重点放在简单卷甚至临时存

注 1：Jeremy Cole. The physical structure of records in InnoDB.

储上将简化管理并减少开销。

1. B-Tree 结构

大多数数据库以二叉树格式来构造数据，也称 B-Tree。B-Tree 是一种数据结构，在保持数据顺序的同时进行自我平衡。B-Tree 已针对数据块的读写进行了优化，这就是 B-Tree 在数据库和文件系统中很常用的原因。

可以将表或索引的 B-Tree 想象成自上而下的一棵树。树有一个根页，根页是基于键所建立的索引的起始点，键是一列或多列。大多数关系型数据库中的表按主键存储，可以显式或隐式地定义主键，例如主键可以是一个整数。如果应用程序寻找特定 ID 或一个 ID 范围对应的数据，则可使用该主键进行查找。除了主键的 B-Tree，还可以定义其他列或列集上的辅助索引。与原始的 B-Tree 不同，这些索引仅存储被索引的数据，而不存储整行。这意味着这类索引要小得多，因而更容易存储在内存中。

B-Tree 称为树，是因为当遍历树时，可以从两个或多个子页中获取所需的数据。如前所述，页包含行数据和元数据，其中元数据包含指向下方页的指针，这些页称为**子页**。根页下有两个或多个页，这些页称为**子节点**。子页或节点可以是内部节点或叶节点，内部节点存储基准键（pivot key）和指向子页的指针，通过指针可以找到另一个节点。叶节点包含键数据。这种结构创建了一棵自平衡树，并且减少了搜索次数，仅进行几次磁盘搜索即可找到所需的数据行。如果所需的数据是键本身，甚至不需要找到该行。

2. B-Tree 写入

将数据插入 B-Tree 时，可以通过搜索找到正确的叶节点。创建节点时会留有富余空间以供插入其他数据。如果节点有空间，则将数据按顺序直接插入节点中；如果节点已满，则必须进行分裂。分裂时，确定新的中位数并创建新的节点，并相应地重新分配数据；然后，将此中间值的数据插入父节点，这可能导致进一步分裂，直到到达根节点为止。更新和删除也是通过搜索找到正确的叶节点，然后进行更新或删除。如果更新的数据多到溢出节点，则可能导致分裂，删除也可能导致重新平衡。

新创建的数据库开始时主要进行顺序读写，故延时较低。随着数据库的增长，分裂会导致 I/O 变得随机，读写延时从而变高。所以在测试过程中必须采用真实数据集，以确保长期运行时的性能特征，而不是运行早期的表现。

单行写入至少需要完全重写一页，如果有分裂，则可能要写入许多页。这种复杂的操作需要原子性，但是如果发生崩溃，则可能导致数据损坏和孤立页。在评估数据存储时，需要了解采用何种机制以避免这种情况。此类机制包括：

- 在复杂的操作落盘之前先写操作日志，也称预写日志；
- 用于重建的事件日志；
- 数据变更前后的重做日志。

考虑到所有这些因素，在配置数据库的底层存储时，关键的一点是数据库块的大小。前面讨论了将数据库块大小与磁盘块大小对齐的重要性，但这还不够。如果使用固态硬盘，则可能发现较小的块在遍历 B-Tree 时性能更优。与机械硬盘相比，在块较大的情况下，固态硬盘的延时可能会增加 30%~40%。由于 B-Tree 结构需要读写，因此必须考虑到这一点。

B-Tree 的属性和优点总结如下。

- 高效的范围查询。
- 不是单行查找最理想的模型。
- 键按顺序存储，以支持高效的查找和范围扫描。
- 该结构可以最大限度地减少大型数据集的页面读取。
- 数据页中不包含键，删除和插入非常高效，仅偶尔需要分裂和合并。
- 如果整个结构可以放进内存，则性能更优。

给数据建立索引还有其他选项，其中最主要的是散列索引。

如前所述，B-Tree 在关系型数据库中非常普遍。如果使用过关系型数据库，则可能已经接触过 B-Tree 了；但还有其他一些选项，并且这些选项正趋于成熟。接下来看看仅追加日志结构[2]。

10.1.2　SSTable和LSM树

BigTable、Cassandra、RocksDB（MongoDB 支持，MySQL 中可通过 MyRocks 引擎开启）和 LevelDB 都采用 SSTable 作为主要存储。术语 SSTable 和 Memtable 最早出现在谷歌的论文“Bigtable: A Distributed Storage System for Structured Data”中，启发了许多 DBMS。

SSTable 存储引擎中有许多文件，每个文件的内部都是一组有序的键值对。与前面讨论的块存储不同，SSTable 在块或行级别无须元数据开销。键及其值对于 DBMS 是不透明的，并且存储为任意的 BLOB。由于它们以有序方式存储，因此可以顺序读取，并将其视为排序所依据键上索引。

有种算法可以在 SSTable 存储引擎中组合内存中的表、批量落盘和定期压缩，该算法称为 LSM（log-structured merge，日志结构合并）树架构，参见图 10-2。

注 2：Goetz Graefe, Harumi A. Kuno. Modern B-tree techniques, 2011.

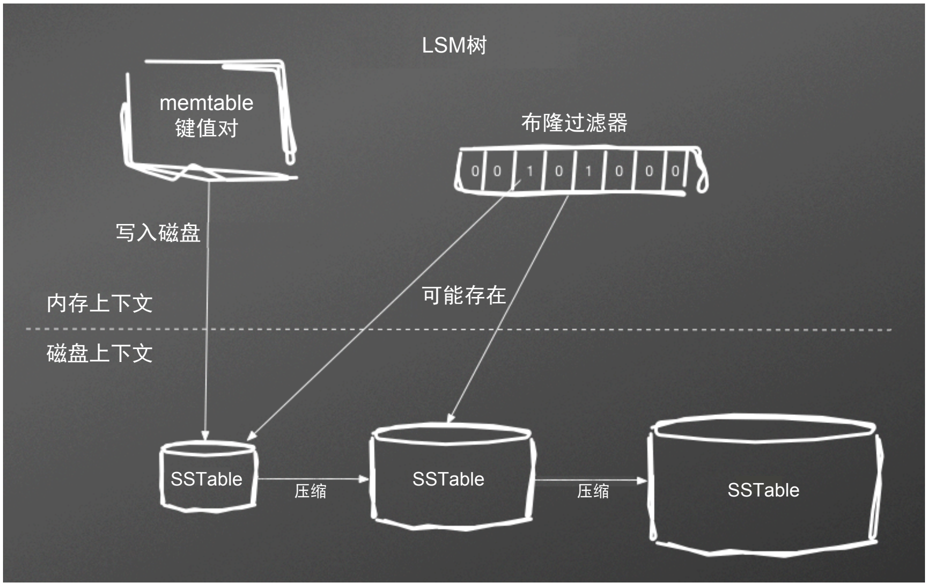


图 10-2：LSM 树结构和布隆过滤器

LSM 树可以定期将内存中的数据写入 SSTable。数据刷新、排序并写入磁盘之后，就是不可变的。不能添加或删除 SSTable 文件中的键值对，这种机制对于只读数据集非常有效，因为可以将 SSTable 加载到内存中实现快速访问。即使 SSTable 不能完全放入内存，也可以最大限度地减少随机读取所需的磁盘寻道。

为了支持快速写入，还需要更多机制。与磁盘写操作不同，内存中数据集的写操作很简单，因为只需更改指针。内存中的表可以在写的同时保持平衡，这称为 memtable。读操作首先查询 memtable，之后读取磁盘上最新的 SSTable，然后查找下一个，直到找到数据为止。达到一定的阈值（可以是时间、事务数或大小）后，将对 memtable 进行排序并将其写入磁盘。

删除已经存储在 SSTable 中的数据时，必须标记为逻辑删除，这称为**墓碑**。 SSTable 会定期合并，消除“墓碑”以节省空间。此合并和整理过程可能会占用大量的磁盘 I/O，并且通常占用更多可用空间。在运维团队习惯这些新容量模型之前，这可能会影响可用性 SLO。

在发生故障时，必须假设存数据可能丢失。在将 memtable 写入磁盘之前，数据是脆弱的。毋庸置疑，SSTable 存储引擎中的解决方案与基于 B-Tree 的存储引擎类似，包括事件日志、重做日志和预写日志。

1. 布隆过滤器

你可能会想，必须搜索 memtable 和大量的 SSTable 来查找不存在的记录，这样做可能既缓慢，成本又高，事实也确实如此。解决此类问题的方案是**布隆过滤器**。布隆过滤器是一种数据结构，可用于评估给定集合中是否存在记录的键，在这种情况下集合就是 SSTable。

像 Cassandra 之类的数据库，使用布隆过滤器来判断哪个 SSTable（若有）可能包含请求的键。布隆过滤器的设计注重速度，因此可能会有一些误报，但是总体上大大减少了读取 I/O。相反，如果布隆过滤器发现 SSTable 中不存在相应的键，则肯定不存在。当 memtable 写入磁盘时将更新布隆过滤器。分配给过滤器的内存越多，误报的可能性就越小。

2. 实现

许多数据库利用 LSM 结构和 SSTable 作为存储引擎：

- Apache Cassandra
- 谷歌 Bigtable
- HBase
- LevelDB
- Lucene
- Riak
- RocksDB
- WiredTiger

每种数据库的实现细节有所不同，但此类存储引擎的激增和日趋成熟已使其成为重要的存储实现方式，任何使用大型数据集的团队都应该理解。

在查看和枚举数据存储的结构时，多次提到了日志对于在发生故障时保证数据持久性的重要性。第 7 章谈到了日志对于分布式数据存储中的数据复制也至关重要。下面深入介绍日志及其在复制中的用法。

10.1.3 索引

前面讨论了最普遍的索引结构之一——B-tree。SSTable 也有内在索引，当然，数据库中也存在其他索引结构。

1. 散列索引

最简单的索引实现之一是**散列表**。散列表是存储桶的集合，桶中包含键的散列函数的运算结果，该散列指向可以找到记录的位置。因为散列表的范围扫描成本过高，所以仅适用于单键查找。另外，散列必须放在内存中以确保高性能。散列表为所适用的特定用例提供了极佳的访问性能。

2. 位图索引

位图（bitmap）索引将其数据存储为位数组（位图）。遍历位图索引时通过按位执行逻辑运算来完成。在 B-tree 索引中，索引在不常重复的值上表现最佳，这也称**高基数**（high cardinality）。当索引的值较少时，位图索引的性能更优。

3. B-tree 索引分类

传统的 B-tree 索引存在子分类，这些分类通常是针对非常具体的用例设计的，如下所示。

- **基于函数**

  索引基于函数的结果。

- **反向索引**

  从值的末尾到开头构建索引，以反向排序。

- **聚簇索引**

  要求表中的记录按索引顺序物理存储以优化 I/O 访问。聚簇索引的叶节点包含数据页。

- **空间索引**

  有多种机制可用于索引空间数据。标准索引类型无法有效地处理空间查询。

- **搜索索引**

  此类索引能够搜索列中的数据子集。大多数索引无法在值内搜索，但一些索引是为此而设计的，并为此操作构建了完整的数据存储，例如 Elasticsearch。

每种数据存储各有一系列专用索引，这些索引通常针对特定用例做了优化。

索引对于快速访问数据子集至关重要。在评估前沿数据存储时，了解索引方面的限制，例如具有多个索引的能力、可以索引多少列甚至后台如何维护这些索引，是至关重要的。

10.1.4　日志和数据库

日志最初是一种维护数据库系统持久性的方法。由于可用性和扩展性的原因，日志逐渐演变成一种将数据从主服务器复制到从服务器的机制。最终，服务被构建成通过转换层，利用日志在不同数据库引擎之间迁移数据。随后日志演变为消息传递系统的一部分——事件，订阅者可以利用事件为下游服务做精细的工作。

日志应用广泛，本章主要关注复制。了解了如何在本地服务器上存储数据和建立索引之后，下面介绍如何将数据复制到其他服务器上。

10.2 数据复制

本书假设数据库可靠性工程师主要处理分布式数据存储。这意味着，必须把一个节点上的数据移动到其他节点。关于这一主题[3]已有很多书了，因此本书着重于介绍众所周知且实用的示例，而不做理论性的讲解。本书旨在帮助数据库可靠性工程师及其所支持的工程师了解复制方法及其工作原理。了解复制相关选项的优缺点及其模式和反模式，对于数据库可靠性工程师、架构师、软件工程师和运维人员至关重要。

复制架构有一些高层次的区别。在讨论复制时，leader 指的是从应用程序中执行写操作的节点，而 follower 是接收复制事件以复制数据的节点，读取节点是应用程序从中读取数据的节点。

- 单 leader

  数据总是发送给特定的 leader。

- 多 leader

  多个节点具有 leader 角色，并且每个 leader 都必须在整个集群中持久化数据。

- 无 leader

  所有节点都能进行写操作。

首先讨论最简单的单 leader 复制，并基于此延伸。

10.2.1 单leader复制

在单 leader 复制模型中，所有写操作由单个 leader 完成并从 leader 中复制数据。因此，在 *N* 个节点中有一个节点被指定为 leader，其他节点都是副本，数据从 leader 中拉取。这种方法因其简单性而应用广泛。这种模型可以保证一些特性，包括：

- 因为所有写入由单个节点完成，所以不会发生一致性冲突；
- 假设所有操作都是确定性的，则可以保证每个节点上的结果都相同。

这里包含一些场景，例如一个 leader 复制到几个副本时写入才被视作成功。无论如何，只有一个 leader 执行写操作，这是该架构的关键属性。在单 leader 模型中有几种复制方法，每种方法都在一定程度上权衡一致性、延迟和可用性。因此，需要根据应用程序的需求以及使用数据库集群的方式选用不同的复制方法。

1. 复制模型

以单 leader 方式复制数据时，可以使用 3 种模型。

注 3：Replication Techniques in Distributed Systems, 1996.

- ❑ **异步**

  通过弱化持久性减少延迟。

- ❑ **同步**

  通过增加延时增强持久性。

- ❑ **半同步**

  延时与持久性之间的权衡。

在**异步复制**模型中，事务写入 leader 的日志，然后提交并写入磁盘；有一个单独的进程将这些日志同步至 follower，follower 会尽快应用这些日志。在异步复制模型中，leader 和 follower 的提交存在一个很小的时间间隙。此外，也不能保证所有 follower 的提交点是相同的。在实践中，提交点之间的时间间隙可能小到可以忽略，有的异步复制系统中 leader 和 follower 的间隙可能达到几秒、几分钟甚至几个小时。

在**同步复制**模型中，事务被写入 leader 的日志，同时通过网络发送给 follower。leader 在收到 follower 确认写操作已经被记录后才会提交该事务。这可以保证集群中的所有节点处于相同的提交点。这意味着读一致性（无论从哪个节点读取），并且如果当前 leader 发生故障，任何节点都可以成为 leader，而没有数据丢失的风险。但网络延时或节点性能下降都会增加 leader 写操作的延时。

同步复制对延时有明显影响，在节点较多的情况下尤其明显，而**半同步复制**是一种折中方案。使用半同步复制算法，只需要一个节点向 leader 确认写操作已被记录，这样在集群中存在性能下降节点的情况下，降低了延时的风险，同时保证了集群中至少有两个节点的提交点是完全一致的。在该模型中，如果从任意节点读数据，并不能保证集群中所有节点会返回相同的数据；但能保证在需要时，至少有一个节点可以在不造成数据丢失的情况下成为新 leader。

2. 复制日志格式

实现单 leader 复制必须使用事务日志。实现这种日志的方法有很多，每种方法各有长短。许多数据存储可能实现了多种事务日志，以便根据需要选择最有效的方式。列举如下。

- ❑ **基于语句的日志**

  在基于语句的复制模式中，写操作对应的 SQL 或语句被记录下来，并从 leader 传输给 follower。这意味着每一个 follower 都会执行整条语句。

  其优点如下。

  - 一条 SQL 语句可以更改成百上千条记录。传输这些受影响的记录需要传输大量数据，而 SQL 语句通常小很多。对于带宽不足的数据中心之间进行复制而言，该方案可能更有效。

- 这种方式易于移植。即使在不同版本的数据库中，大部分 SQL 语句的执行结果也是相同的，这样就可以在升级 leader 之前先升级 follower。这是在生产环境升级时，保证高可用性的重要措施。如果没有向后兼容的复制方式，当整个集群升级版本时，宕机时间会很长。
- 日志文件还可以用于审计和数据集成，因为日志中包含完整的语句。

其缺点如下。

- 如果写入的数据是在选定数据集上使用聚合函数或计算函数算出的，则处理时间可能很长。执行该语句比简单地修改磁盘上的记录更耗时，可能会因串行 apply 过程而导致复制延迟。
- 有些语句的执行结果并不确定，在不同节点上执行时，结果可能不同。

MySQL 基于语句的复制就是这种方式的一个例子。

确定性事务

确定性意味着一条语句的执行结果与时间无关，并且不受外部因素影响。如果一条语句在相同的数据集上以相同的顺序执行，那么无论在哪个节点上执行，结果都应该相同。非确定性语句的例子包括本地时间相关函数，比如 `now()`、`sysdate()` 或随机排序，例如根据 `rand()` 返回值进行排序。

类似地，存储代码（比如自定义函数、存储过程和触发器）可能会使得语句的执行结果变得不确定，从而导致基于语句的复制模式不安全。

- ❑ **预写日志**

预写日志也称重做日志，包含一系列事件，每个事件对应一个事务或者写操作。预写日志中包含应用事务之后磁盘中更改的所有字节。在使用该方式的数据库系统（比如 PostgreSQL）中，日志被直接传输给 follower，该日志用于将数据写入磁盘。

其优点如下。

- 由于语句已经解析并执行，因此操作非常快。需要做的仅仅是将变更落盘。
- 不存在非确定性 SQL 语句的风险。

其缺点如下。

- 在高频写入环境中，会占用大量带宽。
- 由于数据格式和数据库引擎紧密相关，因此不便于移植。因而在滚动升级中，难以减少宕机时间。
- 不便于审计。

持久化和复制机制中使用的预写日志通常是相同的，因此这种方式高效，但可移植性和灵活性不佳。

- ❑ **基于行的复制**

在基于行的复制（也称逻辑复制）中，写操作以事件的形式写入 leader 的复制日志中，该事件指明了行的更新方式、列的新数据、列更新前后的信息以及删除的行。follower 节点可以直接利用该数据修改指定行，而无须执行原来的语句。

其优点如下。

- 不存在非确定性 SQL 语句的风险。
- 复制速度介于前两种算法之间，仍然需要将逻辑转化为实际操作，但不需要执行整条语句。
- 可移植性也介于前两种算法之间。不是非常易读，但仍然可以用于集成和检查。

其缺点如下。

- 在高频写入环境中，会占用大量带宽。
- 不便于审计。

这种方法也称**变更数据捕获**，SQL Server 和 MySQL 支持这种方式。它也用于数据仓库的场景。

- ❑ **块复制**

前面讨论了数据库原生的复制机制。相比之下，块设备复制是解决该问题的外部方法。一种流行的实现方式是 Linux 的 DRBD（Distributed Replicated Block Device，分布式复制块设备）。DRBD 软件在块设备上运行，不仅将数据写入本地的块设备，也复制到其他节点的块设备中。

块复制是同步的，并极大地消除了复制开销。但是，由于不能在第 2 个节点上同时运行数据库实例，因此当进行故障转移时，必须新启动一个数据库实例。如果前一个主库发生故障时没有干净地关闭数据库，新启动的实例就需要恢复数据，这与在同一个节点上重启实例需要恢复数据是相同的。

因此，利用块复制可以实现低延迟的同步复制，但会失去利用副本进行扩展和负载均衡的能力。好在外部复制方法（比如块复制）可以和原生复制方法（比如基于语句的复制、基于行的复制）相结合，从而得到无数据丢失且具有异步复制灵活性的组合方案。

- ❑ **其他方法**

还有其他与数据库日志解耦的复制方法。提取、转换和加载作业在服务之间移动数据，这些作业通常会寻找新行或变更行的标记（比如 ID、时间戳），基于这些标记就可以拉取数据并在别处加载。

表上的触发器也可以记录表的变更，供外部进程监听。这些触发器可能会简单地列出变更的 ID，也可能像基于行的复制方法一样，收集全部变更的数据。

在评估复制方案时，需要将源数据存储、目标存储以及它们之间的基础设施结合起来，稍后会继续讨论。

3. 单 leader 复制方式应用

对于成熟的数据存储而言，复制通常不是必需的，而是可选项；但是，依然有各种各样的原因需要执行复制，而复制方式会影响架构和配置。在单 leader 架构中，原因可以归类为可用性、扩展性、局部性和可移植性。

- **可用性**

  毋庸置疑，当数据库 leader 发生故障时，要有一个最快的恢复选项，以便将应用程序流量指向该恢复选项。拥有一个具有最新数据副本的实时数据库，好于必须先恢复再前滚到故障点的备份。这意味着在选择复制方式时，必须将 MTTR 和数据丢失放在首位。同步复制和半同步复制不会丢失数据且 MTTR 较短，但会影响应用的延时。在没有外部支持（例如除了数据存储还可以将数据写入消息系统，以恢复在异步复制环境中 leader 故障转移过程中可能丢失的数据）的情况下，无法找到仅通过复制就能实现 MTTR 短、延时低且不会丢失数据的方案。

- **扩展性**

  集群写 I/O 受到单个 leader 写入性能的限制，而 follower 可以扩展集群数据的读取能力，读取性能的提升取决于节点数量。对于数据写入相对较少但读取密集的应用，多副本方案确实可以提高集群的吞吐上限。由于复制的开销不是线性增长的，因此这种扩容是受限的。为了支持扩展性，副本上的数据必须足够新以满足业务需求。对于某些组织而言，异步复制系统固有的复制滞后是可以接受的；而对于某些需求而言，无论写延时的影响如何，都必须采用同步复制。

- **局部性**

  利用复制可以将数据集存放在离用户更近的多个位置，进而减少延时。如果用户分布在不同的国家和地区，甚至是不同大洲，那么距离会对请求产生重大影响。大数据集不方便整体迁移，但增量修改可以使数据保持最新状态。如前所述，对于带宽不足的长距离复制，如果数据压缩的大小不足以支撑基于行或预写日志的复制方式，那么通常采用基于语句的复制。当代的网络和压缩技术在一定程度上缓解了这种情况。在延时方面，在长距离下半同步复制和同步复制是不可行的，因而只能选择异步复制。

- **可移植性**

  当前保存在 leader 中的数据，很可能需要存储到其他数据存储中。可以将日志作为数据流水线中消费的事件，推送到数据仓库中，或者转移到其他具有良好查询性能或索引的数据存储中。采用和副本相同的复制流（为了可用性和扩展），可以确保来自 leader

的数据集是相同的。话虽如此，定制方案（比如基于查询的ETL、基于触发器的方法）可以获取经过筛选的合适数据子集，而无须复制日志中的整个事务流。这些作业对数据即时性的要求不高，这使得可以选择对延时影响更小的方法。

基于这些需求，可以选择一种或者多种复制方式。无论怎么选择，在这些复制环境中都会碰到很多挑战。

4. 单 leader 复制方式面临的挑战

在任何复制环境中，都可能遇到很多困难或挑战。虽然单 leader 的复制环境是最简单的，但并不表示它很容易。下面介绍最常见的挑战。

- **构建副本**

  对于大型数据集，数据的可移植性会明显降低，第 7 章讨论过这一点。随着数据集规模的增大，MTTR 会随之增加。为了将 MTTR 限制在可接受的范围内，需要为大规模的副本或新的备份制定策略。其他选项还包括将整个数据集分成多个小数据集，以减少一组服务器中的数据集大小，这种方式也称**分片**（sharding）。第 12 章将进一步讨论。

- **保持副本同步**

  构建副本只是复制环节的第一步。在采用异步复制且频繁或大量更改数据的环境中，保持副本及时同步颇具挑战。如前所述，必须记录更改、发送日志并应用更改。

  关系型数据库的设计通常会将写入转换成一系列严格的串行事务，以确保副本和 leader 之间数据集的一致性。在副本上，通常需要串行执行，一次只能执行一个变更。副本的这种串行 apply 过程往往导致其无法跟上 leader 的数据更新速度，原因如下。

  - 相比 leader 缺少并发和并行，副本的 I/O 资源通常没有得到充分利用。
  - 如果没有相同的读请求，要读取的数据块就可能不在副本的内存中。
  - 如果将读写分离（写 leader，读副本），则并发读请求会影响副本的写延时。

  不管什么原因，最终结果通常称为**副本滞后**。在某些环境中，副本滞后是一种不常见且短暂的问题，通常可以在 SLO 内自行解决。而在另一些环境中，此类问题很常见，并且会导致副本不符合最初的目标。如果发生这种情况，表明数据存储的负载过大，必须通过一种或多种技术重新分布负载。第 12 章将详细讨论此类技术，这里简述如下。

  - **短期**

    增加集群容量，以满足当前负载需求。

  - **中期**

    按数据库的功能拆分为独立集群，以确保集群容量满足工作负载需求。这种方法也称**功能分区**或**分片**。

- **中长期**

  将整个数据集划分到多个集群，以便将工作负载控制在集群容量内。这种方法也称**数据集分区**或**分片**。

- **长期**

  选择一种 DBMS，其存储、一致性和持久性满足工作负载和 SLO，并且不存在扩容问题。

上述可选方案表明，所有方案都无法支撑持续增长，换言之，不能随着负载的增长而线性增长。在应对容量增长方面，一些方案（比如功能分区）比其他方案（比如数据集分区）更容易出现瓶颈；但是，即使是数据集分区方案也有极限。这意味着必须评估其他方面，以确保不会越界（收益递减）而导致解决方案失效。

如果已经出现复制滞后，并且在实施长期方案之前必须缓解由此产生的影响，那么可以采取一些短期策略，包括：

- 将活跃副本的数据集预加载到内存中，以减少磁盘 I/O；
- 降低副本的持久性要求，以减少写延时；
- 在没有跨 schema 事务的情况下，基于 schema 进行并行复制。

这些都是为了争取喘息时间而采取的短期策略，代价是脆弱、维护成本高或者有潜在的数据问题，所以在采用这些策略前必须仔细检查，仅在非常必要时才实施。

❑ 单 leader 故障转移

复制的最大价值之一是存在另外的数据集，并且其中的数据没有滞后，因而在发生故障或需要将流量迁出 leader 时可以充当新 leader。然而，这种操作并不简单，需要很多步骤。在有计划的故障转移中，包括以下步骤。

- 确定将哪个副本作为新 leader。
- 根据不同的拓扑关系，可以变更集群的部分配置，以便让所有副本从新 leader 复制数据。
- 如果是异步复制，则暂停应用程序流量，以便新 leader 更新到最新状态。
- 重新配置应用程序的客户端，让其指向新 leader。

如果编写脚本并自动执行恢复步骤，那么在干净、计划好的故障转移中，这将非常简单；但是在发生故障的情况下，依靠此类自动化的故障转移可能会导致一些问题。计划外的故障转移过程如下。

1. 数据库 leader 实例变得无响应。
2. 心跳监控进程尝试连接数据库的 leader 节点。
3. 心跳监控程序挂起 30 s 后就会触发故障转移。

4. 故障转移算法执行如下步骤：

- 将具有最新提交的副本节点标记为候选 leader 节点；
- 在日志流的合适位置，让其他副本开始从候选节点复制数据；
- 监控集群状态，直到所有副本的数据和候选节点同步完成；
- 通过文件或服务更新集群配置并下发；
- 重建一个新的副本节点。

在此过程中有许多地方容易出错，第 12 章将进一步讨论。

尽管面临这些挑战，复制仍然是数据库普遍具备的重要功能，并已成为数据库基础设施的重要组成部分，这意味着必须把复制纳入可靠性基础设施。

5. 单 leader 复制的监控

有效地监控和运维可见性是有效管理复制的基础，这需要收集和展示很多度量值，以确保副本可以有效地支撑组织的 SLO。必须监控的关键指标包括：

- 复制滞后；
- 复制对写延时的影响；
- 副本可用性；
- 复制一致性；
- 运维流程。

第 4 章讨论过这些内容，但值得在此重提。

❑ 复制的滞后与延时

要想理解复制数据流，首先必须理解执行复制操作所需的相对时间。在异步复制环境中，这意味着在主节点上执行的写操作与在副本节点上执行写操作的时间间隔。该间隔可能会随着时间而变化，但该数据至关重要。有多种方法可以测量该时间间隔。

和其他分布式系统一样，这些度量值在一定程度上依赖本机时间。如果节点的系统时钟或 NTP 发生偏移，就会导致度量值扭曲。绝对不能简单地依赖两台机器的本地时间，并假定时间是同步的。对于大部分采用异步复制的分布式数据库系统而言，这不是问题。两个时间非常接近，这就足够了。但即便如此，记住对于不同节点而言时间是一个相对概念，将有助于分析问题。

要测量数据从插入 leader 到插入副本节点的耗时，一种常用的方法是插入一行心跳数据，然后测量它出现在副本节点上的时间。例如在 12:00:00 插入一条数据，并定期轮询副本，如果发现副本没有收到该数据，则可以假定复制已停止；如果在 12:01:00 查询时发现 11:59:00 的数据存在，但 12:00:00 及以后的数据不存在，则可知在 12:01:00 时复制滞后 1 s，之后可以使用下一行来度量数据库当前的滞后情况。

对于半同步复制或完全同步复制，需要了解复制配置对写入的影响。这将作为总体延时度量的一部分。除此之外，还需要度量从 leader 到副本节点网络传输的耗时，因为这是通过网络进行同步写操作的耗时。

以下是必须收集的关键度量值：

- 在异步复制中，leader 节点与副本节点之间数据同步的延时；
- leader 节点和副本节点之间的网络延时；
- 同步复制对写延时的影响。

对任何服务而言，这些度量值都是很重要的。代理可以利用复制滞后信息确定哪个数据库副本的数据是及时更新的，可以承担生产环境的读流量。代理层也可以移除副本节点，以保证不会读到陈旧的数据，落后的副本也无须承担读负载，从而可以更快地同步到最新状态。当然，该算法必须考虑所有副本都出现延时的场景：由 leader 节点提供服务？从前端截断流量直到所有副本同步更新完成？将系统设为只读模式？这些都是应对这种场景的可能选项。

除此之外，工程师还可以利用副本的滞后与延时信息，排查数据一致性、性能下降或其他由复制延时导致的问题。

下面看看下一组度量值：可用性和容量。

❑ 复制的可用性和容量

如果使用的是基于副本因子进行数据同步的数据库（例如 Cassandra），还需要监控满足 quorum 读的可用副本数。假设集群的副本因子为 3，这表示一次写入会复制到 3 个节点。应用程序要求在查询时，此次写入数据的节点中有三分之二的节点返回结果。如果有两个节点发生故障，将无法满足应用程序的查询需求。主动监控副本可用性可以预知故障风险。

类似地，即使在没有副本因子和 quorum 要求的环境中，为了符合 SLO，数据库集群仍需要能监控有多少节点可用。监控当前集群中正常工作的节点数是否满足这些要求是至关重要的。

最后，需要及时发现复制中断。虽然采用心跳方式监控复制滞后，会通知复制落后，但如果发生异常并且复制已经中断，心跳监控并不会报警。有很多因素可能导致复制中断。

- 网络分区。
- 在基于语句的复制中，无法执行 DML，包括：
  - schema 不匹配；
  - 非确定性 SQL 语句导致数据集漂移，从而违反约束条件；
  - 意外到达副本的写操作导致数据集漂移。
- 权限 / 安全设置变更。

- 副本空间不足。
- 副本数据损坏。

需要收集的度量值示例如下。

- 实际可用的数据副本与期望副本数。
- 需要修复的副本（复制中断的）。
- leader 节点和副本节点之间的网络度量值。
- 数据库 schema、用户以及权限的变更日志。
- 复制日志消耗的存储。
- 数据库日志，可以提供关于复制错误和数据损坏的相关信息。

有了这些信息，自动化程序就能够根据副本的可用性度量值，在副本不足的情况下部署新副本。运维人员也能快速查明故障的根本原因，以便决定是修复副本还是直接替换，抑或有更多系统问题待解决。

❑ 复制的一致性

如前所述，有些场景会导致 leader 节点与副本节点的数据集不一致。有时，如果这种不一致导致 apply 阶段的复制失败，就会因复制中断而收到报警。更糟糕的是，长时间未被发现的静默数据损坏。

第 7 章讨论了验证流水线对于维护数据集、业务规则和约束一致性的重要性，类似的流水线能够确保副本之间的数据是一致的。与检查一致性的数据验证流水线一样，从资源角度看，这通常既不简单，成本也不低，所以必须有选择性地确定检查对象以及检查的频率。

只追加写的数据结构（例如 SSTable），或只新增数据的表（B-Tree 结构）易于管理。这是因为可以基于主键或日期范围在一组行上创建校验和，并在副本之间比较这些校验和。只要运行得足够频繁且及时，就能确保数据是相对一致的。

对于允许修改的数据，可能更具挑战性。一种方法是在应用程序执行事务之后，对数据应用数据库级散列函数（当将其合并到复制流中时）。如果数据复制正常，每个副本中的散列值是相同的；如果复制不正常，散列值将不同。比较近期事务散列值的异步任务可以发现是否存在数据差异。

以上仅介绍了可以监控复制一致性的几种方法。为软件工程师创建范式并分类数据对象，以帮助他们决定是否需要执行验证流水线，这样可以减少不必要的资源浪费。基于存储的数据类型[4]来决定采用采样验证还是时间窗口验证，也是一种有效的方法。

注 4：Todd Anderson, Yuri Breitbart, Henry F. Korth, et al. Replication, Consistency, and Practicality: Are These Mutually Exclusive?

- ❑ **运维流程**

  最后，监控运维流程所需的时间和资源对于复制至关重要。随着时间的推移，数据集和并发都在增长，这些流程在很多方面变得越来越繁重。如果超过一定阈值，复制的数据可能无法保持最新，或者无法为了支撑流量而维持一定数量的副本。这些度量值包括：

  - 数据集大小；
  - 备份持续时间；
  - 副本恢复时长；
  - 备份和恢复过程所占用的网络流量；
  - 恢复后的同步时间；
  - 备份时对生产节点的影响。

  每次备份、恢复或者同步时，发送带有合适度量值的事件来创建报告，以便评估或者预测数据集大小和并发量何时会导致运维流程不可用。可以根据操作持续的时间或者消耗的资源，随着数据集和并发量的变化规律，进行一些基本的预测性评估。

  在自动化预测之外定期检查和测试，有助于运维人员评估操作流程何时不再可扩展，以便提前准备更多容量、重新设计系统或流程，或重新平衡数据集的分布，来高效地支持可用性和延时 SLO。

  虽然在数据复制方面不可避免地会有其他度量值和指标，但一套有效的监控指标可确保的复制有效地工作并符合 SLO。

由于相对简单，单 leader 复制是迄今为止最常用的复制方式。不过，有时这种方法无法满足可用性和本地化需求。通过允许从多个 leader 节点写入数据，可以减轻 leader 节点故障转移带来的影响，并且可以将 leader 节点分布于不同的区域来提升性能。下面看看该要求的方法和挑战。

10.2.2 多leader复制

打破单 leader 复制范式实际上有两种方法。第 1 种方法名为“多向复制”，或者“传统多 leader”。该方法中，仍然存在 leader 角色的概念，并且所有 leader 都被设计成可以写入数据，并将数据传到副本以及其他 leader。通常有两个分布在不同数据中心的 leader。第 2 种方法支持从任意节点写入，即数据库集群中的任意节点随时都可以有效地处理读写请求，写入的数据随后会传到其他节点。

这里无论尝试哪种解决方案，最终结果都很复杂，因为需要增加一层用于解决冲突。当所有写请求都到单个 leader 时，有一个前提——不同节点之间的写是不会冲突的；但是如果允许写入多个节点，就可能发生冲突，必须有合适的手段来解决冲突，这就增加了应用程序的复杂性。

1. 多 leader 用例

如果最终多 leader 的复制机制很复杂，那么值得为哪些需求承担其成本和风险呢？

- **可用性**

  在单 leader 异步复制中，当 leader 节点进行故障转移时，应用程序通常会受到至少 30 s 的影响，有时影响会持续 30 分钟甚至数小时（取决于系统设计），这是由于需要进行数据一致性检查、崩溃恢复或其他步骤。

  在某些情况下，这种服务中断可能是不可接受的，并且没有资源或者能力通过修改应用程序来更透明地容忍故障转移。在这种情况下，可能值得为写操作跨节点负载均衡的能力接受不可避免的复杂性。

- **本地化**

  为了向全球或者分布在不同地区的用户保证低延时，业务可能需要在不同区域运行。在读密集型应用程序中，通常可以通过长距离的单 leader 复制来解决。然而，如果应用程序是写密集型的，长距离发送写请求的延时可能会非常高。在这种情况下，在每个数据中心都部署一个 leader 并解决冲突是更好的办法。

- **灾难恢复**

  与局部性和可用性相似，有时应用程序非常重要，必须在多个数据中心分别部署，以便在个别数据中心偶发故障的情况下保证可用性。这仍然可以通过单 leader 复制来实现，但前提是仅将辅助区域用于只读（如前所述）或者冗余。但是，很少有企业能够负担得起不让整个数据中心“物尽其用”。因此，通常选择多 leader 复制让两个数据中心都可以向用户提供服务和支持。

随着在云服务中运行的基础设施占比的增长或者全球分布的需求，最终几乎不可避免要评估多 leader 复制。多 leader 复制的实现通常可以获得原生支持，也可以通过第三方软件实现。挑战是如何处理不可避免的冲突。

2. 传统多向复制中的冲突解决

传统的多向复制与单 leader 最为相似。本质上，当多个 leader 允许写请求时，写请求就会转发给所有 leader。这听起来不错，并且可以满足前面讨论的所有用例。但是，如果采用异步复制（这是在包含多个数据中心和网络连接缓慢的环境中唯一可行的方法），则可能出现问题。在复制延迟或网络分区期间，依赖数据库的应用程序将遇到陈旧数据。在修复复制滞后或网络分区时，必须解决写入数据版本不同的问题。数据库可靠性工程师和软件工程师如何处理多 leader 复制体系中的写冲突问题呢？他们需要非常小心。有多种方法可以解决该问题。

- **消除冲突**

  最简单的解决方式是避免发生冲突。有时可以通过某种方式控制写操作或请求来避免冲突，示例如下。

- 为每个 leader 分配一个主键子集，其主键只能在该特定 leader 上生成，这对于仅插入 / 追加的应用程序非常有效。最简单的方式是，一个 leader 负责奇数主键，另一个 leader 负责偶数主键。
- 亲和力方法，始终将特定用户路由到特定 leader。可以按地区、唯一 ID 或其他方式操作。
- 仅将第 2 个 leader 用于故障转移，实际上一次仅写入一个 leader，但维护多 leader 拓扑以便进行故障转移。
- 在应用程序层分片，在每个区域中部署完整的应用程序栈，从而消除跨区域复制。

当然，以这种方式进行配置并不意味着一直是有效的。配置错误、负载均衡错误和人为错误都可能发生，并且可能导致复制中断或数据损坏。因此，即使冲突只是偶尔发生，也需要做好准备。如前所述，错误越罕见，危害可能越大。

❑ 最后写入有效

对于无法避免的潜在写冲突，需要确定在发生冲突时如何处理。数据存储中原生提供的常见算法之一是 LWW（last write wins，最后写入有效）。在 LWW 中，当两个写操作发生冲突时，具有最新时间戳的写操作是有效的。这似乎很简单，但是时间戳有很多问题。

时间戳——甜蜜的谎言

大多数服务器时钟使用挂钟时间，该时间取决于 `gettimeofday()` 函数的返回结果，该数据由硬件和 NTP 提供。许多因素会导致时间倒流而不是前进，例如：

- 硬件问题；
- 虚拟化问题；
- 未启用 NTP，或者上游服务器可能出错；
- 闰秒。

闰秒相当棘手。POSIX 把一天定义为 86 400 s，但真正的一天并不总是如此。闰秒通过跳过或重复计算秒数使各天保持一致。这可能会导致严重的问题，谷歌把闰秒均匀分布在一天中，以保证时间的均匀性[5]。

有时 LWW 相对安全。如果能确定写数据的正确状态，可以执行不可变的写操作，则可以使用 LWW。但是，如果依赖已在事务中读取的状态来执行写操作，则在网络分区的情况下很可能会丢失数据。

Cassandra 和 Riak 是采用 LWW 的数据存储示例。在 Giuseppe DeCandia 的论文“Dynamo: amazon’s highly available key-value store”中，LWW 是其描述的处理更新冲突的两个选项之一。

注 5：参见“Time in Distributed Systems”。

- **自定义解决冲突的选项**

  由于依赖时间戳的算法有其局限性，因此通常需要考虑更多自定义选项。当检测到写冲突时，许多复制器能够执行自定义代码。自动解决写冲突所需的逻辑可能非常广泛，但即使如此，也可能犯错。

  使用乐观复制（允许写入和复制所有变更），可以让后台进程、应用程序甚至用户确定解决这些冲突的方法。这就像选择数据对象的一个版本一样简单。也可以考虑完全合并数据。

- **无冲突的复制数据类型**

  解决冲突的自定义代码逻辑很复杂，因此许多组织可能不愿承担相应的工作与风险。然而有一类数据结构可以有效地管理来自多个副本的写入，这些写入可能存在时间戳或网络方面的问题，这类数据结构称为 CRDT（conflict free, replicated datatype，无冲突的复制数据类型）。CRDT 提供了强大的最终一致性，因为它们始终能在没有冲突的情况下进行合并或解决。在撰写本文时，CRDT 已应用于 Riak 以及大型在线聊天软件中。

  由此可知，在多 leader 环境中解决冲突是可能的，但并不简单。分布式系统的复杂性是非常现实的，需要大量工程时间和精力。此外，当前的数据存储中可能没有实际可用的成熟实现。所以，在涉水多 leader 复制之前，一定要非常小心。

3. 任意节点写入的复制

传统的多向复制有另一种范式。在“任意节点写入”的方法中没有 leader，任何节点都可以读写，例如 Riak、Cassandra 和 Voldemort 等基于 Dynamo 的系统就采用了这种复制方法。下面详细介绍这些系统的一些属性：

- 最终一致性；
- 读写 quorum；
- 非严格的 quorum；
- 反熵。

不同的系统在实现上会有所区别，但这些系统都采用无 leader 复制的方式。只要应用程序可以容忍无序的写入，就可以使用此类系统。通常可以通过一些可调参数调整此类系统的行为，以更好地满足需求，但是无序写入还是不可避免的。

- **最终一致性**

  “最终一致性”一词经常被认为与名为 NoSQL 的数据存储有关。在分布式系统中，服务器或网络可能发生故障。此类系统设计成分布式是为了实现持续的可用性，但有损数据一致性。在节点关闭数分钟、数小时甚至数天的情况下，不同节点存储的数据很容易产生差异[6]。

注 6：Werner Vogels. Eventually Consistent.

当系统恢复正常时，将利用前文介绍的方法加以解决，包括：

- 利用时间戳或矢量时钟的 LWW[7]；
- 自定义代码；
- CRDT。

尽管不能保证任何时候所有节点上的数据都是一致的，但是这些数据最终会收敛。在构建数据存储时，可以配置必须写入多少个数据副本才能在故障期间进行判定。

话虽如此，仍然必须证明最终一致性是可行的。无论是对冲突解决技术的错误理解，还是应用程序的 bug，都可能导致数据丢失。Jepsen 是一个很棒的测试套件，可以有效地测试分布式数据存储中数据的完整性。其他一些阅读材料包括：

- Jepsen 的文章“Distributed Systems Safety Research”；
- Martin Fowler 的文章“Eventual Consistency”；
- Peter Bailis 和 Ali Ghodsi 的文章“Eventual Consistency Today: Limitations, Extensions, and Beyond”。

❑ 读写 quorum

任意节点写入复制的一个关键因素是，必须了解读写操作保证一致性的最少节点数。在客户端或数据库级别，通常可以定义 quorum。从历史上看，quorum 是小组开展业务所需的最少成员数。对于分布式系统，这是保证数据一致性所需的最少写入节点或读取节点数。

例如，在由 3 个节点组成的集群中，能容忍一个节点发生故障，这意味着至少需要两个读取节点和两个写入节点。在确定所需节点时，有一个简单的公式，其中 N 是集群中的节点数量，R 是读取节点的数量，W 是写入节点的数量。如果 $R+W>N$，则达到一个有效的 quorum 数，可以保证写入后至少有一个节点的读取是正确的。

在 3 个节点的示例中，这意味着给定 2+2>3，至少需要两个读取节点和两个写入节点。如果两个节点发生故障，则只有 1+1=2 个节点（<3），因此没有达到 quorum，集群不应在读取时返回数据。如果在读取两个节点时应用程序收到不同的结果（一个节点上丢失数据或数据不同），则使用相应的冲突解决方法进行修复，这称为**读修复**。

关于 quorum 以及分布式系统的相关理论和实践，还有更多内容，推荐以下阅读材料：

- Moni Naor 和 Avishai Wool 的论文“The Load, Capacity, and Availability of Quorum Systems”；
- Marko Vukolić 的 *Quorum Systems: With Applications to Storage and Consensus*。

注 7：Roberto Baldoni, Michel Raynal. Fundamentals of Distributed Computing: A Practical Tour of VectorClock Systems.

- ❑ **非严格的 quorum**

  有时部分节点正常运行，但不能满足 quorum 的需求。可能 *N*1、*N*2 和 *N*3 被配置为提供写操作，而 *N*2 和 *N*3 处于关闭状态，但 *N*1、*N*4 和 *N*5 可用。此时，系统应禁止对应数据的写操作，直到重新将节点加入集群中并恢复 quorum 为止。但是，如果接收写请求更重要，则可以通过非严格的 quorum 提供写操作，这种情况下其他节点可以临时处理写请求以满足 quorum。一旦 *N*2 或 *N*3 在集群中启动成功，就可以通过名为 hinted hand-off 的过程将数据回传给 *N*2 和 *N*3。

  quorum 是一致性和可用性之间的权衡。了解数据存储实际上如何实现 quorum 至关重要。必须了解何时允许非严格的 quorum，以及哪些 quorum 可以保证强一致性。文档可能会产生误导，因此必须测试实现的实际情况。

- ❑ **反熵**

  保持最终一致性的另一个工具是反熵。在读修复和 hinted hand-off 之间，基于 Dynamo 的数据存储可以非常有效地保持最终一致性。但是，如果不经常读取数据，则不一致可能会持续很长时间。如果将来进行故障转移，应用程序可能收到的是陈旧数据。因此，需要一种额外的机制来同步数据，该过程称为反熵。

  反熵的一个示例是 Merkle 树，可见于 Riak、Cassandra 和 Voldemort 的实现中。Merkle 树是数据对象散列的平衡树。通过构建分层树，反熵的后台进程可以快速识别节点之间的不同值并进行修复。这些散列树在写入时会被修改，并会定期清除和重新生成，以最大限度地降低丢失不一致数据的风险。

  对于存储大量冷数据的数据存储，反熵至关重要，它是对 hinted hand-off 和读修复的良好补充。确保这些数据存储实现了反熵的功能，将有助于在分布式数据存储中最大限度地保证一致性。

尽管这些系统的实现细节存在显著差异，但组成元素就是前面讨论的组件。假设应用程序可以容忍无序的写入和陈旧的读取，则无 leader 复制系统可以提供出色的容错性和扩展性。

了解 3 种常用的数据存储复制方法，有助于从高层次理解跨系统分布数据所采用的方法。可以基于团队的经验和专长以及对可用性、规模、性能和数据局部性的需求，设计出符合组织需求的系统。

10.3 小结

本章是数据存储的速成课程，介绍了如何将数据存储在磁盘上，以及如何围绕集群和数据中心推送数据。这些内容是数据库架构的基础。接下来将深入研究数据存储的属性，以帮助你和团队选择适合组织需求的架构。

第 11 章

数据存储领域指南

从技术上讲，数据存储就是数据的存储以及用于数据存储、访问和修改的相关软件及结构。本书主要讨论当今组织用于满足大量用户高并发访问大量数据需求的数据存储。

传统上，读者借助领域指南识别植物、动物或其他自然物体，即区分相似物体。本章旨在介绍各种数据存储的特征，这些信息有助于你理解这些数据存储的最佳用例及注意事项。

本章首先定义数据存储的属性和类别，这些与读写数据的应用程序开发人员密切相关，然后深入研究架构师和运维人员最感兴趣的数据存储类别。尽管我们认为，开发人员、设计人员和运维人员都应该了解数据存储的所有属性，但必须承认，人们往往从各自工作的角度评估这些属性。市面上数据存储产品的种类很多，本书无意一一介绍，而会通过示例讲解数据存储并提供分析工具，以便读者根据自己的需要和目标深入研究。

11.1 数据存储的概念属性

数据存储的分类方法有很多。实际上，如何分类取决于具体的工作内容以及与数据存储的交互方式。应用程序中是否有查询、存储和修改数据的需求？是否为决策目的而查询和分析数据？是否设计了数据库运行系统？是否管理、调优或监控数据库？不同的角色对数据库和其中数据有独特的认识。

在 ORM 或 Serverless 等开放 API 的架构中，已经出现了将数据存储从其使用者中抽象出来的趋向。我们并不看好这种趋向。理解所选择的数据存储的每个属性及其含义，对于做好工作至关重要。天下没有免费的午餐，每个有吸引力的功能都伴随着其他特性的割舍。使用这些数据存储的团队需要充分理解这一点。

11.1.1 数据模型

对于大多数软件工程师而言，数据模型是最重要的分类之一。如何构造数据以及如何管理数据关系，对于在其上构建的应用程序而言至关重要，也会极大地影响管理数据库变更的方式，因为不同模型管理变更的方式千差万别。

下面介绍 4 种常见的数据模型：关系模型、键值模型、文档模型和导航模型（图模型），每种模型的用途、限制和特性都不同。关系模型历来最为流行。由于在生产中进行了大规模实践，因此关系模型更容易理解、更稳定、风险更小。

1. 关系模型

关系模型最初由 E. F. Codd 在 IBM 内部提出，并于一年后的 1970 年发表了论文“A Relational Model of Data for Large Shared Data Banks”。由于本指南的目的不是介绍完整的背景知识，而是帮助读者理解当今常见的系统，因此将重点介绍现代组织中普遍采用的关系模型系统。

关系型数据库模型的基本前提是把数据表示为一系列关系，这些关系基于唯一键（数据的核心标识符）。关系模型通过对关系、基数、值和某些属性的约束，在不同表之间确保数据一致性。关系模型是固定的，包括各种严格的约束，也称**范式**。实际情况是，由于性能和并发问题，理论上的许多需求未完整实现 [1]。

著名的关系型数据库有 Oracle、MySQL、PostgreSQL、DB2、SQL Server 和 Sybase。该领域的其他数据库有谷歌 Spanner、亚马逊 RedShift、NuoDB 和 Firebird。其中一些数据库系统被归类为 NewSQL，被视作关系型数据库管理系统的子类，在保持一致性的同时试图突破并发和可伸缩的瓶颈，稍后进一步讨论 [2]。

关系模型提供了一种非常著名的数据检索方式。通过连接以及一对多和多对多关系，开发人员可以灵活地定义数据模型。这也可能导致具有挑战性的 schema 演变，因为表、关系和属性的添加、修改和删除都需要大量协调和变更。如第 8 章所述，这可能导致成本高昂且有风险的变更。

许多软件团队选择使用 ORM 层，通过将关系模型映射到软件层定义的对象模型来简化工作。此类 ORM 框架对于提升开发速度而言是很好的工具，但是对于数据库可靠性工程师团队而言，会带来诸多问题。

注 1：E. F. Codd. The relational model for database management. 2nd ed. 1990.

注 2：E. F. Codd. The relational model for database management. 2nd ed. 1990.

> **ORM 与你**
>
> 过去十年里，ORM 已经发展成熟，数据库可靠性工程师无须像以前那样警惕，但仍需考虑一些问题。
>
> - ORM 对表进行读写操作，使得对部分工作负载的任何优化都具挑战性，因为会影响整体工作负载。
> - ORM 持有事务的时间可能比需要的更长，因而会对有限的资源造成重大影响，因为快照被过度维护。
> - ORM 可能导致大量不必要的查询。
> - ORM 可能导致复杂且性能不佳的查询。
>
> 除了这些明显的问题，更大的挑战包括 ORM 将数据库抽象出来，消除了组织扩张（超过为其工作的数据库管理员的数量）时所需的协作 ORM 还会导致忽略约束、混淆逻辑，并妨碍数据库可靠性工程师理解应用程序与数据存储之间的交互[3]。

这些问题使得许多软件工程师和架构师认为关系型数据库不够灵活，并且影响了开发速度。这样的评价并不准确，稍后将列出一份更准确的利弊列表，并打破列表中许多流行的“神话”。

2. 键值模型

键值模型将数据存储为字典或散列表。字典与表类似，包含任意数量的对象，每个对象可以存储任意数量的属性或字段。与关系型数据库类似，这些记录也由唯一的键标识，但无法根据这些键创建对象之间的映射。

键值数据存储将对象视为数据块，但并不知道其中的数据，因此每个对象可以有不同的字段、嵌套对象以及无限的多样性。这种多样性是有代价的，例如可能导致数据不一致，这是由于在公共存储层没有强制规则。类似地，也没有高效的数据类型和索引。键值数据存储消除了管理各种数据类型、约束和关系的大量开销。如果应用程序不需要这些特性，则可以提高效率。

键值存储的例子有很多，比如 Dynamo。2007 年亚马逊发表了关于 Dynamo 的论文，介绍了如何构建高可用的分布式数据存储。当讲解完所有属性，我们将详细介绍 Dynamo。基于 Dynamo 的系统包括 Aerospike、Cassandra、Riak 和 Voldemort。其他键值实现还包括 Redis、Oracle NoSQL 数据库和 Tokyo Cabinet。

3. 文档模型

从技术上讲，文档模型是键值模型的子集。文档模型的不同之处在于，数据库维护关于文

注 3：Christopher Ireland, David Bowers, Michael Newton. et. al. A Classification of Object-Relational Impedance Mismatch.

档结构的元数据，允许进行数据类型优化、辅助索引以及其他优化。文档存储将与对象相关的所有信息存储在一起，而不是跨表存储。这样一次调用便能获取所有数据，而无须表连接（声称简单，实际会消耗大量资源）。通常也不需要 ORM 层。

另外，这意味着如果对象需要不同的视图，那么文档存储必然不需要范式，这会导致膨胀和一致性问题。此外，由于没有 schema 这样的简明文档，因此需要使用外部工具来实施数据治理[4]。

数据治理

数据治理是对组织数据的可用性、完整性和安全性的管理。应该仔细考虑并记录新添加的数据属性。在数据存储中使用 JSON，会很容易甚至意外添加新的数据属性。

4. 导航模型

导航模型始于分层和网络数据库。如今提到导航模型，几乎总是指图数据模型。图数据库使用节点、边和属性来表示存储的数据以及对象之间的连接。节点包含特定对象的数据，边是对象之间的关系，属性可添加节点相关数据。因为关系是作为数据的一部分直接存储的，所以易于追踪连接。通常可以在一次调用中检索整个图。

与文档存储一样，图存储通常能直接映射到面向对象应用程序的结构中，同时消除了表连接，并且在数据模型演化方面更灵活。当然，这只适用于查询图的数据。实践证明传统查询模式的性能可能差得多[5]。

每种模型都有适用场景。下面讨论其中的选择和权衡。首先看看事务和属性。

11.1.2 事务

数据存储如何处理事务，也是需要考虑和理解的重要属性。事务实际上是数据库的逻辑工作单元，可以认为不可分割。事务中的所有操作必须全部执行或回滚，以保持数据存储的一致性。确保整个事务都将被提交或回滚，将极大地简化应用程序的错误处理逻辑。事务模型的此类保证能够让开发人员忽略某些故障和并发性，否则会耗费大量时间和资源。

如果主要使用传统的关系型数据存储，可能认为事务的存在是理所当然的。这是因为这些数据存储几乎都是建立在 ACID 模型（稍后介绍）上的，该模型由 IBM 于 1975 年提出。所有读写都被视作事务，并利用数据库底层的并发架构来实现。

注 4：Harley Vera, Wagner Boaventura, Maristela Holanda, et al. Data Modeling for NoSQL Document-Oriented Databases.

注 5：Michael Stonebraker, Gerald Held. NETWORKS, HIERARCHIES AND RELATIONS IN DATA BASE MANAGEMENT SYSTERMS.

1. ACID

ACID 数据库提供了多项保证，分别是：atomicity（原子性）、consistency（一致性）、isolation（隔离性）和 durability（持久性）。1983 年，Theo Härder 和 Andreas Reuter 提出了该缩略词。ACID 基于 Jim Gray 提出的原子性、一致性和持久性（缺少隔离性）。这 4 个特性描述了事务范式的主要保证，它影响了数据库系统开发的许多方面。

在使用数据存储时，需要了解它是如何定义和实现这些概念的，因为可能存在歧义和多样性。鉴于此，必须考虑每一种特性，并了解在现实中可能出现的变化[6]。

2. 原子性

原子性是指保证将整个事务提交到（或写入）数据存储，或回滚整个事务。支持原子性的数据库不存在部分写或部分回滚。这里的原子性不同于软件工程中的原子操作，后者指的是并发隔离（这样其他进程能看到进行中的工作，而不是只看到操作前的状况和操作后的结果）。

事务执行失败并需要回滚的原因有很多。客户端进程在事务进行到一半时可能终止，或网络故障导致连接中断。类似地，数据库崩溃、服务器故障和其他许多操作可能需要回滚部分完成的事务。

PostgreSQL 通过 `pg_log` 实现事务，事务写入 `pg_log`，并设置“进行中”“提交”或“中止”状态。如果客户端放弃或回滚一个事务，则标记为中止。如果没有任何后端映射到事务，后台进程也会定期将事务标记为中止。

需要注意，只有在底层磁盘页的写操作为原子性的情况下，才可以考虑写操作的原子性。对于扇区写操作的原子性方面人们存在重大分歧，大多数现代磁盘为写扇区引入电源，即使在磁盘故障期间也是如此。但是，取决于物理驱动器和落盘之间的抽象，仍然存在丢失数据的可能。

3. 一致性

一致性是保证任何事务都将数据库从一个有效状态转换到另一个有效状态。可以假定事务不能违反已定义的规则。从技术上讲，一致性是在应用程序级别而不是数据库级别定义的。然而，传统数据库为开发人员提供工具来保证一致性。这些工具十分高效，包括约束和触发器。约束包括级联外键、非空、唯一性约束、数据类型和长度，甚至是特定字段中允许的值。

既有趣又困扰的是，一致性这个术语也用于数据库和软件领域的其他方面。CAP 定理也用了这个术语，但含义相去较远。在讨论散列和复制时，也会涉及该术语。

4. 隔离性

隔离性是一种承诺，即并发执行事务与串行和顺序执行这些事务的最终状态是相同的。

注 6：Marco Vieira, António Casimiro Costa, Henrique Madeira. Timely ACID Transactions in DBMS.

支持 ACID 的数据库通过写锁、读锁和快照的技术组合来实现这一点，统称**并发控制**。实际上，有多种并发控制可能导致数据库的行为不同。严格的并发控制会显著影响并发事务的性能，而宽松的并发控制以减少隔离换取更好的性能[7]。

ANSI/ISO SQL 标准定义了 4 种可能的事务隔离级别，相同的事务在不同的级别下结果可能不同。这是根据每个隔离级别是否允许 3 种潜在事件来定义的。

- **脏读**
  脏读指可能读取其他客户端事务写入的未提交数据或脏数据。

- **不可重复读**
  不可重复读指在事务的上下文中如果执行两次相同的读，则由于数据库中其他并发活动可能导致结果不同。

- **幻读**
  幻读指在事务的上下文中，执行两次相同的读，第 2 次返回的数据与第 1 次的不同。与不可重复读不同，在幻读中，已经查询的数据不会改变，但是返回的数据多于以前。

为了避免上述现象，可以采用以下 4 种隔离级别。

- **读未提交**
  这是最低的隔离级别。在该隔离级别下，存在脏读、脏写、不可重复读和幻读。

- **读已提交**
  在该隔离级别，目标是避免脏读和脏写。换言之，不能读取或覆盖未提交的数据。一些数据库通过在选定的数据上获取写锁来避免脏写，持有写锁直到数据提交，读锁在查询之后就释放。脏读通常是由于写入数据的多个副本导致的，读到了部分已经提交的数据和部分未提交的数据。

  在读已提交隔离级别，仍会遇到不可重复读。如果之前读取了数据，提交之后再次读取，则在自己的事务上下文中会出现不同的值。

- **可重复读**
  为了实现读已提交隔离级别并避免不可重复读，必须采用额外的控制机制。如果数据库使用锁控制并发，则客户端需要保持读写锁直到事务结束。不过，它并不会持有范围锁，因此可能出现幻读。可以想见，这种基于锁的方法非常繁重，可能会显著影响系统高并发的性能。

  另一种实现此目的的方法是快照隔离。在快照隔离中，事务启动后，客户端将看到当前数据库的镜像。获取快照后的写操作不会出现在快照中，这使得长时间运行的读操作具

注 7：Atul Adya, Barbara Liskov, Patrick O’Neil. Generalized Isolation Level Definitions.

有一致、可重复的查询结果。快照隔离使用写锁而不是读锁，旨在确保读操作不会阻塞写操作，反之亦然。由于存在两个以上副本，因此这称为**多版本并发控制**（multiversion concurrency control，MVCC）。

在可重复读快照隔离中，仍可能发生写偏序。在写偏序中，两个写操作可能发生在同一列上，也可能发生在同一行的不同列上，因为已经读取了正在更新的列，所以将导致行中的数据来自两个事务。

- **串行化**

  这是最高级别的隔离，旨在避免上述所有现象。与可重复读类似，如果锁是并发控制的焦点，那么需要在事务期间持有读锁和写锁，但还需要添加一些锁定策略，因此称为两阶段锁定（2-phase locking，2PL）。

  在 2PL 中，锁分为共享锁和排他锁。多个读操作可以共享读锁，但必须在其他事务释放所有共享读锁之后，才能获得写操作的排他锁。类似地，如果正在进行写操作，则无法获取读操作的共享锁。在这种模式下，在高并发环境中多个事务因为相互等待锁而被阻塞是很常见的，这种现象称为**死锁**。此外，还必须在 `WHERE` 子句中使用范围查询获取范围锁，否则将发生幻读。

  2PL 会极大地影响事务的延时。当大量事务处于等待状态，整个系统的延时都将大增。所以，很多系统只实现了可重复读，而非全面的可串行性。

  非基于锁的方法基于快照隔离，称为**串行快照隔离**（serial snapshot isolation，SSI）。这是一种乐观串行化的方法，在该方法中，数据库将等待提交，以检查是否存在导致串行性问题（通常是写冲突）的事务。在并发冲突很少的系统中，这样做明显减少了延时；但是，如果经常发生冲突，那么经常回滚和重试带来的问题可能非常明显。

因为每个隔离级别都比其下的隔离级别更严格，所以在较低隔离级别禁止的行为在高隔离级别仍被禁止。该标准使得 DBMS 能在更高（比所需的隔离级别）的隔离级别上执行事务（例如“读已提交”事务可以在“可重复读取”隔离级别上执行）。

隔离的不同实现

如前所述，在执行 ANSI 隔离标准时，数据存储之间存在显著差异。

- PostgreSQL：有读已提交、可重复读和串行化的隔离级别，串行化采用 SSI。
- Oracle：只有读已提交和串行化的隔离级别，该串行化比实际的串行化更接近可重复读。
- MySQL/InnoDB：有读已提交、可重复读和串行化的隔离级别，串行化使用 2PL，但不检测更新的丢失[8]。

注 8：参见 Baron Schwartz 的博文“If Eventual Consistency Seems Hard, Wait Till You Try MVCC”。

前面只讨论了隔离、隔离异常和隔离实现的皮毛。章末将推荐一些有趣的读物，供深入阅读。

5. 持久性

持久性是一种保证：一旦事务被提交，就永远提交了，无论是否存在断电、数据库崩溃、硬件故障或其他任何问题，事务都将持久有效。显然，数据库不能保证底层硬件支持这种持久性。如第 5 章所述，数据库很可能以为数据已经同步到磁盘，实际情况却并非如此。

持久性与原子性密切相关，因为原子性需要持久性。许多数据库实现了预写日志，以便在写操作同步到磁盘之前捕获所有写操作，该日志用于事务撤销或重放。如果发生故障，在重新启动时，数据库可以检查该日志，以确定是否需要撤销、完成或忽略事务。

就像隔离级别一样，有时为了性能可适当放松持久性。为了真正的持久性，每次提交时都必须刷新磁盘。这样做成本可能较高，并不是所有事务和写操作都需要这样做。例如在 MySQL 中，可以把 InnoDB 日志刷新调整为定期执行，而不是每次提交后都执行。也可以对复制日志执行该操作[9]。

从更高的层次看，支持事务的系统显然隐藏了许多实现细节。组织中的数据库可靠性工程师应确保除自己外，整个开发团队都熟悉数据库的实现。文档中的这些实现细节通常并不显而易见，但可以使用 Jepsen 和 Hermitage 之类的工具进一步测试来加深理解。

类似地，当有降低持久性或弱隔离的选项时，这些知识有助于选择合适的配置。另外，了解数据库默认值何时不能满足应用程序的需求也同样重要。

11.1.3 BASE

工程师们在寻找传统关系型系统的替代方案时，已开始把 BASE 一词用作 ACID 的同义词。BASE 代表**基本可用**（basically available）、**软状态**（soft state）以及**最终一致性**（eventual consistency）。BASE 重点关注非事务性分布式系统，这些系统的复制和同步机制可能与传统数据库不同。与 ACID 系统不同，当此类系统正常运行并接收流量时，可能永远不会处于明确的状态。同样，无须事务的并发控制，以牺牲原子性、隔离性和一致性为代价，可以大幅提高读写吞吐量和并发性[10]。

研究过数据库的数据模型和事务模型之后，下面介绍与开发人员最相关的概念。在评估数据库选型以及数据库周边的整个运维生态系统和基础设施时，还必须考虑其他许多属性（见表 11-1）。

注 9：Russell Sears, Eric Brewer. Segment-Based Recovery: Write-ahead loggin revisited.

注 10：Charles Roe. The Question of Database Transaction Processing: An ACID, Base, NoSQL Prime.

表11-1：数据存储概念属性汇总

属性	MySQL	Cassandra	MongoDB	Neo4J
数据模型	关系	键值	文档	导航
模型成熟度	成熟	2008	2007	2010
对象关系	外键	无	DBRef	模型核心
原子性	支持	分区级别	文档级别	对象级别
一致性（节点）	支持	不支持	不支持	强一致
一致性（集群）	基于副本	最终一致（可调节）	最终一致	XA 事务支持
隔离性	MVCC	串行化选项	读未提交	读已提交
持久性	DML 支持，DDL 不支持	支持（可调节）	支持（可调节）	支持，预写日志

很多事情被过度简化了。质疑特性的真实表现以及测试这些功能，有助于在为应用程序评估数据存储时明晰问题。即使有这些警告，仍然存在一些明显差异，这可用于为应用程序做出合适的选择。下一步是评估数据存储的内部属性以获得全面认识。

11.2　数据存储的内部属性

有多种方法可以描述和分类数据库。数据模型和事务结构是直接影响应用程序架构和逻辑的属性，因此追求速度和灵活性的开发人员会重点关注这两个属性。数据库的内部架构实现往往是黑盒，或者只是营销手册上吹嘘的功能。尽管如此，它们对于选择合适的长久数据存储至关重要。

11.2.1　存储

第 10 章详细介绍了存储。每种数据存储都有一个或多个将数据写入磁盘的选项，通常形式为存储引擎。存储引擎管理数据的读写、锁、数据的并发访问以及管理数据结构所需的任何流程，例如 B-tree 索引、LSM 树和布隆过滤器。

某些数据库（例如 MySQL 和 MongoDB）提供了多个存储引擎选项，例如在 MongoDB 中，可以使用 MMap、WiredTiger 以及 LSM 树或基于 LSM 树的 RocksDB。存储引擎的实现各有不同，但是存储引擎的属性通常可以细分为以下内容：

- 写性能；
- 读性能；
- 写操作的持久性；
- 存储空间。

根据这些属性评估存储引擎，有助于为数据存储选择合适的存储引擎。通常要在读写性能和持久性之间权衡。还可以通过一些特性来提高存储引擎的持久性。了解这些以及对持久性的真实性进行基准测试是非常重要的。

11.2.2　无处不在的CAP理论

人们在讨论这些属性时，通常指的是 Eric Brewer 的 CAP 定理[11]（见图 11-1）。根据 CAP 定理，任何通过网络共享数据的系统最多可以满足以下三个属性中的两个：一致性（consistency）、可用性（availability）和分区容错性（partition tolerance）。类似于 ACID 中的术语，这些术语过于笼统。每个属性都不是单独的，实际上相互关联。许多系统强调 CP 或 AP，这意味着此类系统为了两个特定属性而牺牲了另一个属性。但是，如果深入研究这些系统，就会发现它们对每个特定属性的实现都是不完整的，仅实现了部分可用性或一致性。

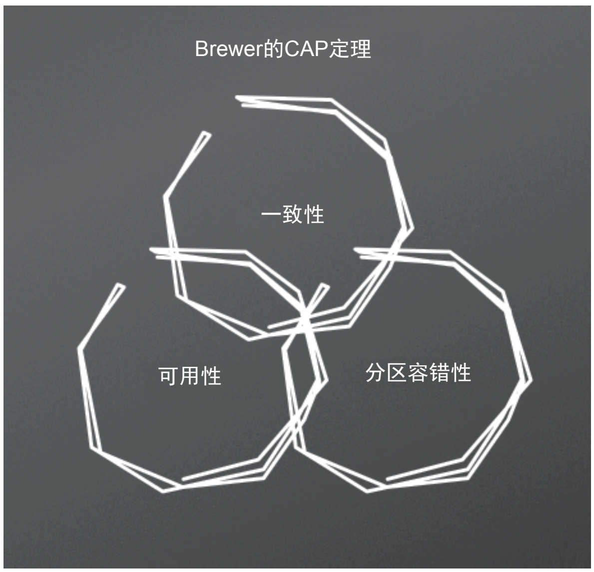


图 11-1：Brewer 的 CAP 定理：一致性、可用性和分区容错性

CAP 旨在帮助设计人员了解一致性和可用性之间的权衡。分布式系统中的网络分区是不可避免的。网络本质上是不可靠的，在这种情况下，如果分区后允许更新节点的状态，则不可避免地会失去一致性。如果需要一致性，则其中一个分区必然不可用。下面仔细研究每个属性及其相关内容。

1. 一致性

前面讨论过一致性。困扰的是，ACID 中的一致性与 CAP 中的一致性不同。在 ACID 中，一致性意味着事务保证所有数据库规则和约束；而在 CAP 中，一致性意味着**线性一致性**。线性一致性保证了对分布式数据库中对象的一组操作将按时间执行。因为操作包含

注 11：Martin Kleppmann. Please stop calling databases CP or AP.

读和写，所以这些操作必须按执行的顺序呈现给系统中的其他用户。线性化是顺序一致性的保证[12]。

在网络分区的情况下，ACID 中的一致性（比如 CAP 一致性）无法保证，这意味着基于 ACID 的事务数据存储在网络分区时只能通过牺牲可用性来保证一致性[13]。因此发展出了 BASE 理论，以便在不牺牲可用性的情况下容忍网络分区。

2. 可用性

CAP 定理的可用性是指处理请求的能力。通常大多数分布式系统可以提供一致性和可用性，但在网络分区的情况下，其中一部分节点与另一部分节点分割开来，所以如果必须保证可用，要以牺牲一致性为代价。当然，没有任何系统可以一直保持完全的可用性，这体现了之前所说的可用性与其他属性是相连的，而不是孤立的属性。

3. 分区容错性

网络分区是连接的暂时或永久中断，最终导致网络基础设施中的两个子集的通信中断。实际上，这样通常会形成两个较小的集群。这两个集群都认为自己是最后存活的集群，可以继续提供写入服务。这会产生两个数据集，这种情况称为**裂脑**。

CAP 定理的提出，是为了帮助人们理解分布式数据存储中一致性和可用性之间的权衡。实际上，网络分区在数据存储的生命周期中只占很少的时间，系统应该同时具备一致性和可用性。但是，当发生分区时，系统必须能够发现并加以管理，以恢复一致性和可用性。

如果 CAP 定理完全不考虑延时或性能，就没有实际价值。延时与可用性同样重要，高延时也是导致一致性问题的潜在原因。过长的延时会迫使系统进入与网络分区相似的故障状态。需要权衡延时，但这种权衡通常比权衡一致性和可用性更为明确。实际上，BASE 系统和 NoSQL 变革兴起的另一个重要原因是对大规模提升性能的需求。

了解了 CAP 定理，下面介绍它如何影响数据库分类。可以采用 CP 与 AP 的概念，但如前所述，这种方法过于简化。首先看看分布式系统如何保持一致性和可用性。

11.2.3　一致性与延时的权衡

在分布式系统中，如果所有节点都以相同的顺序写入事务，则称该系统强一致；换言之，系统是线性化的。CAP 定理专门讨论了在网络分区的情况下，分布式数据存储如何提高一致性或可用性。但是，在数据存储的整个生命周期中都需要一致性，并且不能仅通过 CAP 范式来看待一致性[14]。

注 12：Peter Bailis. Linearizability versus Serializability.

注 13：Eric Brewer. CAP Twelve Years Later: How the “Rules” Have Changed.

注 14：Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design.

大家都希望分布式数据存储具有强一致性，但很少有人愿意接受强一致性对延时和可用性的影响，因此需要权衡。下面讨论权衡取舍以及它们如何影响集群中的整体一致性，以便评估数据存储是否符合需求。

将数据写入分布式数据存储中的节点时，必须复制数据以保证可用性。如前所述，复制数据有多种方式。

- 将写操作同步发送到所有节点。
- 将写操作发送到一个节点（主节点）；异步、半同步或同步复制到其他节点。
- 将写操作发送到任意节点，该节点仅充当该事务的主节点；异步、半同步或同步复制到其他节点。

当集群中的任何节点都可写时，如果没有协调程序（例如 Paxos）有效地对写操作进行排序，则有可能破坏一致性，但排序必然会增加事务的延时。这是保持强一致性同时权衡延时影响的一种方法。如果延时比排序更重要，则在此阶段可能会牺牲一致性。这就是**排序与延时**之间的权衡。

写操作发送到其中一个节点（主节点），然后必须复制到其他节点。由于主节点可能关闭 / 损坏或由于负载过大而导致超时，进而导致接收写操作的主节点不可用。重试或等待会增加延时，但可以配置负载均衡或代理服务器，在超时后将写操作发送到其他节点。但是，这样做可能会导致一致性问题，因为如果原始的事务已被处理但未确认，则会产生冲突。增加重试次数或扩大超时窗口会影响延时，在保证一致性的同时会影响可用性。这是**主节点超时与重试**之间的权衡。

从节点读取数据时，也会遇到超时或不可用的情况。在异步复制的环境中，将读操作发送到其他节点可能会使其读取陈旧数据，从而导致一致性问题。增加超时和重试次数会降低结果不一致的风险，但会增加等待时间。这是**读取超时与重试**之间的权衡。

同步写入所有节点时，无论是通过有序处理还是通过复制，将所有事务发送到其他节点所产生的开销会导致额外的延时。如果这些节点处于拥堵的网络中或跨网络通信，则延时可能会非常高。这是**同步复制与延时**之间的权衡。一种折中方案是半同步复制，通过减少参与节点和网络连接的数量来减少延时。但是，增加延时就增大了数据丢失的风险，从而牺牲了可用性。这是**半同步与可用性和延时**之间的权衡。

上述权衡展示了提高系统一致性或减少延时的方式。在系统发生网络分区时，这些权衡对于保证系统继续提供服务至关重要。

11.2.4 可用性

同一致性及其与延时的关系类似，也需要考量可用性。类似于 CAP 定理，发生网络分区时也需要考虑可用性。但是，面对单节点故障、多节点故障或整个集群故障，每天都会出

现可用性问题。在讨论分布式系统中的可用性时，采用**产量**（yield）和**收成**（harvest）的概念，而不是简单地归结为可用性，是很有帮助的。产量指完成一个请求的可能性，收成指结果数据集的完整性。发生故障时不是简单地考虑正常运行还是宕机，而是简单地评估哪种方法最佳——降低产量或减少收成。

在分布式系统中，问自己的第一个问题是，减少收成以保持产量是否可接受。如果 25% 的节点失效，查询中提供 75% 的数据是否可接受？如果搜索返回大量结果，则可以接受。若是如此，那么可以更好地容忍故障，这可能意味着可以减少 Cassandra 环中的副本因子。同样，如果收成必须保持接近 100%，则需要更多数据副本。这不仅意味着有更多副本，而且还应该有更多可用区来保存副本。

这一点在把应用程序分解为子应用程序时也有体现。无论是功能分区还是微服务中都经常出现这种情况，结果是将故障与系统的其余部分隔离开来。这通常需要编写代码，但这是减少收成以确保产量的例子。

了解存储机制以及数据存储对一致性、可用性和延时的权衡，可以学习到数据存储的“幕后”知识，这是对本书所介绍概念的补充。负责应用程序性能和功能的工程师和架构师更关心概念属性，而运维人员和数据库工程师通常更关注内部属性（参见表 11-2），以确保符合设定的 SLO。

表11-2：数据存储内部属性总结

属性	MySQL	Cassandra	MongoDB	Neo4J
存储引擎	插件，主要是 B-tree	LSM	插件，B-tree 或 LSM	原生图存储
分布式一致性	聚焦一致性	最终一致性，相比可用性较次要	聚焦一致性	聚焦一致性
分布式可用性	相比一致性较次要	聚焦可用性	相比一致性较次要	相比一致性较次要
延时	基于持久性调优	写优化	基于一致性调优	读优化

11.3 小结

本章介绍了市面上各种数据存储的属性。对于考虑新应用程序、研究现有应用程序，或评估开发团队对最新数据存储的需求，它都很有帮助。既然已经从存储进阶到数据存储，自然要讨论数据架构和流水线。

第12章

数据架构示例

前面讨论了存储引擎和单个数据存储，本章拓宽视野，看看这些数据存储如何适应多系统架构。只涉及一种数据存储的架构比较少见，实际中会采用多种方式来保存数据，并且数据有多个消费者和生产者。本章将介绍一个简洁的架构示例，然后介绍流行的数据驱动架构，以及这些架构试图解决的问题。

本章介绍如何有效地使用这些组件，及其对数据服务带来的积极影响或消极影响。虽然不会面面俱到，但可以概览生态系统和所需内容。

12.1 架构组件

架构组件都在数据库可靠性工程师的日常职责范围之内。可以忽略数据生态系统所有组件的时代已成历史，如今所有这些组件都对数据服务的可用性、数据完整性和一致性有一定影响。在设计服务和运维流程时，绝对不能忽略它们。

12.1.1 前端数据库

前端数据库是本书的重要内容。应用程序的用户通常通过数据访问层查询、插入和修改这类数据库中的数据。过去许多应用程序的功能设计并没有考虑前端数据库的可用性。这意味着，一旦前端数据存储宕机或响应慢到影响用户体验，应用程序就会变得不可用。

通常把此类系统称为 OLTP（on-line transactional processing，联机事务处理）系统。其特点是快速处理大量事务，因此是为了保证高速查询、高并发情况下的数据完整性，以及并发事务量的扩展性而设计的。用户希望所有数据都是实时的，并且包含支撑服务的所有细

节。每个用户或事务都在寻找数据的一个子集，这意味着查询模式倾向于在大型集合中查找和访问特定的小数据集。高效的索引、隔离和并发是关键，这就是通常由关系型系统实现的原因。

前端数据存储的另一个特点是，其数据主要是由用户输入的。还有一些面向用户且主要用于分析的数据存储，通常称为 OLAP（on-line analytics processing，联机分析处理），稍后讨论。

前面讨论了大部分数据存储的各种特性：存储结构，数据模型，ACID/BASE 范式，可用性、一致性以及延时之间的权衡。此外，还必须考虑整体的可操作性，以及与生态系统其他部分之间的集成。这方面需要考虑的典型特性有：

- 低延时的写入和查询；
- 高可用；
- MTTR 较短；
- 在线可扩展；
- 易于和应用程序及运维服务集成。

对于任何单一架构而言，要满足这些需求都是非常困难的；在架构中如果没有其他组件的协助，几乎无法满足这些需求，稍后会阐述这一点。

12.1.2　数据访问层

应用程序通常分为展现层和业务逻辑层。数据访问层（data access layer，DAL）位于业务逻辑层。这一层为应用程序访问持久化数据存储提供了简便的方式。这通常表现为一组对象，这些对象包含相关存储过程或查询的属性和方法。这种抽象向软件工程师隐藏了数据存储的复杂性。

数据访问层的一个例子是 DAO。DAO 通过将应用程序调用映射到数据库，来提供数据库访问接口。通过单独存放持久性逻辑，软件工程师可以单独测试数据访问；类似地，通过提供 mock 接口而非数据库，也能测试应用程序。关于这种方法，通常认为需要在 JDBC（Java database connectivity）或同等模块编写更多代码。尽管如此，当需要通过某些方法满足性能需求时，更靠近数据库可以提高编码效率。另一个常被提及的不足是，这种方式要求开发人员深入理解 schema。但我们认为这恰恰是优势，开发人员对该 schema 理解得越深入，对所有参与者越有利。

数据访问层的另一个示例是 ORM。尽管已经明确表示，出于多种原因我们并不喜欢 ORM，但它也有一些长处。ORM 可以提供很多特性，包括缓存和审计。了解软件工程师团队使用了什么，以及在数据访问的编码和优化中引入了哪些灵活性或约束是至关重要的。

12.1.3 数据库代理

数据库代理层位于应用程序服务器和前端数据存储之间。一些代理位于网络传输分层模型的四层（L4），并利用该层的可用信息来决定如何将应用程序服务器的请求分发到数据库服务器。这些信息包括数据包报头中的源、目的 IP 地址以及端口。利用 L4 功能可以根据特定算法分配流量，但无法将负载或复制延时等其他因素考虑在内。

> **四层和七层**
>
> 在本书中，层指的是 OSI（open systems interconnection，开放式系统互联）模型的层。该模型定义了网络的标准。

七层（L7）代理在网络传输分层模型的最高层（应用程序层）运行，在本例中为 HTTP 层。七层代理可以访问 TCP 包中更多的数据，也可以理解数据库协议和协议路由，并且可以高度定制。

其中一些功能如下。

- “健康检查”以及将请求重定向到健康的服务器。
- 读写分离，将读请求发送到副本节点。
- 重写查询，以优化无法在代码中调优的查询。
- 缓存查询结果并返回。
- 将流量重定向到没有滞后的副本节点。
- 生成查询的度量值。
- 对查询类型或主机进行过滤。

当然，上述所有功能都是有代价的，这里的代价指延时。因此，使用四层代理还是七层代理，取决于团队对功能和延时的权衡。通过代理在不同层解决问题，有助于减轻技术债的影响；但是，这可能导致技术债在较长时期内被忽略，从而导致在数据库方面应用程序的可移植性降低。

1. 可用性

代理服务器的主要功能之一，是在节点故障期间将流量重定向。对于副本节点，代理服务器可以进行“健康检查”，并从服务中移除故障节点。在主节点发生故障或者写失败的情况下，如果只有一个写节点，代理服务器就可以停止服务，以便安全地进行故障转移。无论哪种情况，使用代理都可以显著缩短故障的 MTTR。这里假设代理层设置为具有容错性，否则只是新增了一个故障点。

2. 数据完整性

如果代理只是简单地重定向流量，则对数据的完整性几乎没有影响。然而，在有些情况下，代理可能会提高或者降低数据完整性。在异步复制环境中，七层代理可以从服务中移

除任何出现复制滞后的副本节点，这就减少了返回陈旧数据的可能。

另外，如果代理服务器缓存数据以减少延时并提高数据库节点的处理能力，那么若在写操作之后未能使缓存中的数据失效，则有可能从缓存中返回陈旧数据。稍后会讨论该问题以及其他缓存问题。

3. 扩展性

设计良好的代理层可以极大地增强扩展性。前面讨论了多种扩展模式，其中包括将读请求分发到多个副本节点。如果不使用代理，则只能进行基本的负载分配，但这种方式无法感知副本节点的负载及复制滞后情况，因此用处不大。对于读密集型工作负载，利用代理分发读请求是非常有效的方法。其前提是，业务收益可观，可以支撑副本节点的支出，并且已经建立了有效的自动化方法进行管理。

代理层还可以通过减轻负载来增强扩展性。许多数据库服务器遭遇过大量并发连接的情况，在这种场景中，代理层可以充当连接队列并持有大量连接，但只允许一定数量的连接执行数据库操作。尽管并发延时增加了，这似乎是反直觉的，但限制连接的数量和数据库执行的操作可以提高吞吐量。

4. 延时

当在事务流程中添加一层时，延时是重要的考虑因素。四层代理增加的延时较低，而七层代理导致的延时很高。可以通过一些改进方法摊还此类延时。这些改进方法包括缓存经常执行的查询、避免服务器过载以及重写低效的查询。权衡因应用程序的不同而不同，需要由数据库可靠性工程师、架构师和软件工程师一起做出决定。与大多数权衡一样，应简化功能而非丰富功能（除非必要）。简单性和较低的延时对组织非常有价值。

了解了数据访问层和代理层——帮助应用程序访问数据库的层，接下来讨论在数据库下游运行的应用程序，它们是从前端数据库中消费、处理、转换数据并创造价值的系统。

12.1.4　事件与消息系统

数据不是孤立存在的。由于事务发生在主库中，因此在一个事务完成之后，必须执行其他许多操作。换言之，这些事务如同事件。事务发生之后可能采取的操作的示例如下：

- 数据必须进入下游分析程序和数据仓库中；
- 必须履行订单；
- 欺诈检测必须审查交易；
- 数据必须上传到缓存或 CDN 中；
- 个性化选项必须重新校准和发布。

以上是事务完成后可能触发操作的几个示例。构建事件和消息系统是为了利用数据存储中的数据并发布事件，供下游流程处理。消息和事件软件可通过异步消息在应用程序之间共

享数据。这些系统基于在数据存储区中检测到的信息生成消息，其他应用程序会订阅并消费这些消息。

许多应用程序支持消息发布和订阅等功能。在撰写本文时，最受欢迎的是 Apache Kafka，主要用于分布式日志。Kafka 在生产者、消费者和 topic 级别都支持大规模水平扩展。其他系统有 RabbitMQ、亚马逊 Kinesis 和 ActiveMQ。简单而言，这是一个提取、转换和加载过程，持续或定期轮询数据存储中的新数据。

1. 可用性

事件系统可能会给数据存储的可用性带来积极影响。具体而言，将事件及对事件的处理从数据存储中剥离出来，能从数据存储中消除一类工作负载。这会降低资源利用率和并发量，进而影响核心服务的可用性。这也意味着即使在高峰期间也可以处理事件，因为不必担心干扰生产环境。

2. 数据完整性

在不同系统之间移动数据面临的最大风险之一是，可能导致数据损坏和丢失。在有多个数据源和数据消费者的分布式消息总线中，数据验证面临严峻挑战。对于数据不能丢失的情形，消费者必须将某种形式的副本写回到总线中；然后，进行审计的消费者可以读取这些消息并将其与原始消息进行比较。与之前讲过的数据验证流水线一样，这需要大量编码和资源。对于不能丢失的数据而言，这是绝对必要的。当然，对于能够容忍一定程度丢失的数据而言，采样校验是可行的。如果检测到数据丢失，则需要通过某种方法通知下游进程重新处理特定消息，具体措施取决于消费者。

同样，重要的是验证事件或消息的存储机制能否保证消息在生命周期内的持久性。如果数据可能丢失，则存在数据完整性问题，与之相反的是数据重复。如果存在数据重复，则会导致事件被重复消费。如果不能保证处理是幂等的（事件重复消费的最终结果一致），那么最好选用可以通过索引来管理重复数据的数据存储。

3. 扩展性

如前所述，通过将事件及其后续处理从前端数据存储中分离出来，能减少数据库的总体负载。这就是工作负载分区，之前介绍扩展模式时讨论过，这是可扩展路径上的一步。通过解耦工作负载消除了多模式工作负载的干扰。

4. 延时

将事件处理从前端数据存储中分离出来，可以显著减少潜在冲突，进而减少应用程序的延时。但是，将事件从前端数据存储发送到事件处理系统所花费的时间，是处理这些事件的额外延时。该过程的异步特点意味着应用程序必须能容忍处理中的延时。

至此，介绍了如何访问数据存储，以及在前端数据存储和下游消费者之间打通数据，下面看看其中一些下游数据消费者。

12.1.5 缓存和内存存储

与内存相比，磁盘访问速度慢得令人难以置信。这就是为什么应尽量将所有数据集都放入内存（例如缓冲区缓存）中，而不是直接从磁盘读取。话虽如此，对于许多环境而言，没有那么多预算将数据集都缓存到内存中。对于对数据存储的缓存而言太大的数据，不妨考虑使用缓存系统和内存数据存储。

缓存系统和内存数据存储非常相似。它们的功能是将数据存储在 RAM 而非磁盘上，从而提供快速的读取访问。如果数据不常更改，并且可以容忍内存数据存储的易失性，那么这可能是一个极好的选择。许多内存数据存储通过后台进程异步地将数据写入磁盘以保证持久性。但这带来了很大的风险：数据在保存之前，因发生崩溃而丢失。

内存数据存储通常具有其他功能，例如高级数据类型、复制和故障转移。较新的内存存储还针对内存访问做了优化，甚至比关系系统中全部数据集都在缓存中还要快得多。数据库缓存仍然要验证数据是否为最新的，还要管理并发和 ACID 需求。因此，内存数据存储对极低延时的系统来说可能更适合。

有 3 种填充缓存的方法。第 1 种方法是将数据写入关系型数据库之类的持久性数据存储之后，将其放入缓存中。第 2 种方法通过双写同时写入缓存和持久性存储。由于其中可能会有一个失败，因此该方法非常不可靠；想要保证双写的可靠性，需要高昂的开销，包括写后验证和两阶段提交。第 3 种方法是先写缓存，然后将其异步地写入磁盘，也称**直写**（write through）方法。下面讨论每种方法是如何影响数据库生态系统的。

1. 可用性

缓存会给可用性带来积极影响，即使在数据存储发生故障的情况下，仍可以处理读请求。对于读密集型应用程序，这可能非常有价值。此外，尽管缓存系统可以改善容量或延时，但在缓存系统发生故障时，数据存储无法承受相应的流量，因此缓存层的可用性与数据存储的可用性同样重要，这意味着管理的复杂度是原来的两倍。

另一个主要问题是惊群效应（thundering herd）。在惊群效应下，所有缓存服务器都非常频繁地访问某些数据，这是由于写操作或超时导致的。在这种情况下，大量服务器同时将读取请求发送到持久性存储，以便刷新缓存。这可能会导致并发冗余请求，从而造成持久性存储过载，进而可能导致缓存或持久层无法提供读服务。

应对惊群效应有多种方法。一种简单的方法是确保缓存超时彼此偏移，但这种方法不可扩展。增加代理缓存层来限制对数据存储的直接访问，该方法更易于管理。此时，有一个持久层、一个代理缓存层和一个缓存层。显然，扩展性可能很快会变得非常复杂。

2. 数据完整性

在缓存系统中，数据完整性可能是非常棘手的问题。由于缓存的数据通常是不断变化的数

据的某种静态副本，因此必须在数据刷新频率与对持久性存储的影响之间做权衡，来应对可能读取到陈旧数据的情况。

在保存数据之后再存入缓存时，必须应对读取数据可能过期的情形。这种方法适合很少需要失效和重新缓存的相对静态的数据，例如地理编码、应用程序元数据以及只读内容（比如新闻文章或用户生成的内容）。

同时将数据写入持久性存储和缓存，消除了读取过期数据的可能，但仍需要验证数据是否过期。在写操作之后立即（此后定期）进行验证，以确保向消费者提供正确的数据。

最后，当先把数据写入缓存，再写入持久性存储时，必须能够重新构建写请求，以应对在将写请求转发给持久性层之前缓存系统发生崩溃的情况。记录所有写请求，并将其视为可以触发验证代码的事件，是一种可行的方法。这种方法会在架构层面增加复杂性（许多数据存储库内置了相关机制）。直写方式在验证和维护跨数据存储完整性方面带来了复杂性，因此必须仔细考虑这样做是否值得。

3. 扩展性

使用缓存和内存数据存储的主要原因之一是提升工作负载的读性能。因此，添加缓存层是以增加环境复杂性为代价而换取更高的扩展性。但是，如前所述，现在正在建立对该层的依赖关系，以便符合 SLO。如果缓存服务器发生故障、失效或损坏，则持久性存储难以支撑应用程序的读取压力。

4. 延时

除了扩展性，使用缓存层的另一个原因是减少读延时。这是缓存或内存技术的一种典型用法；但是如果缓存服务器发生故障，在没有缓存服务器的情况下，将无法确定持久存储层能否支撑当时的流量。所以，需要定期测试，测试的读流量会绕过测试环境的缓存，以了解持久层如何同时处理读写工作负载。如果生产环境因缓存故障而导致读取持久性存储，就需要在生产环境中测试此类故障场景。

缓存和内存数据存储是许多流行的数据驱动架构的可靠组成部分。它们可以和事件驱动的中间件很好地结合，并增强应用程序的扩展性和性能。话虽如此，在运营支出、故障风险和数据完整性风险方面，又需要多管理一层。这不容忽视且经常发生，因为这是普遍采用的一种方案。数据库可靠性工程师的职责是确保组织认真对待该子系统的可用性和数据完整性。

这些组件在数据存储的可用性、扩展性和功能增强中都起着关键作用。但是，每一项都增加了架构的复杂性、操作依赖性、数据丢失的风险和数据完整性问题。因此，在做架构相关决策时，需要权衡折中。前面研究了一些单一组件，接下来介绍架构，它将数据从前端产品传到数据存储，再到下游服务。

12.2 数据架构

本章中的数据架构是关于数据驱动系统的示例，这些系统接收、处理和传递数据。前面介绍了基本原理、用法和权衡，旨在为本书中一直讨论的数据存储和相关系统提供真实场景。当然，这些仅仅是示例，实际应用会因每个组织的需求而有很大差异。

12.2.1 Lambda和Kappa

Lambda 是一种实时大数据架构，已被许多组织采用。Kappa 是一种响应模式，追求简洁性并采用较新的软件技术。接下来首先介绍基本架构，然后讨论各种组合。

1. Lambda 架构

Lambda 架构用于快速处理大量数据，以满足近实时的请求，还支持长时间运行的计算。Lambda 由 3 层组成：批处理层、实时处理层和查询层，如图 12-1 所示。

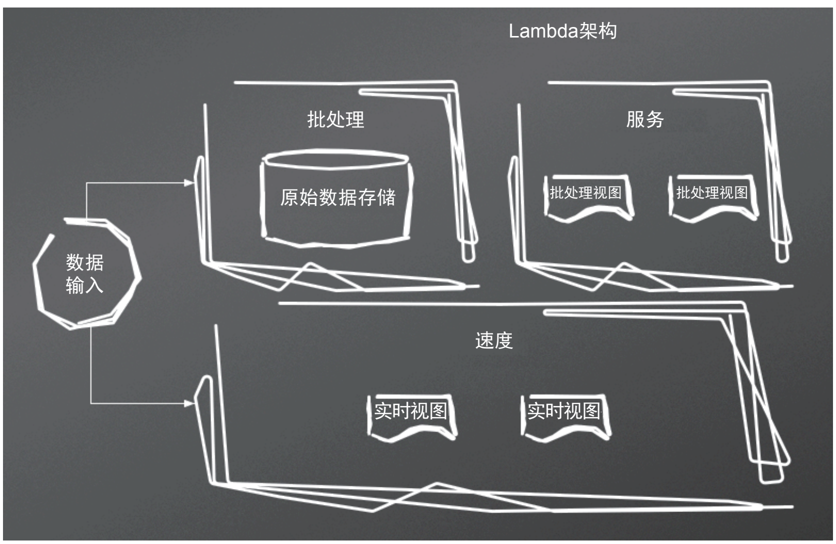


图 12-1：Lambda 架构

如果将数据写入前端数据存储，则可以使用分布式日志系统（例如 Kafka），为 Lambda 处理层生成不可变的分布式日志。一些数据可能不经过数据存储而直接写入日志服务，处理层会处理此类数据。

Lambda 有两个处理层，因此可以快速处理来支持快速查询，还可以进行全面、准确的计算。批处理层通常由 MapReduce 实现，其延时无法满足实时或近实时的查询。批处理层的典型数据存储是分布式文件系统，例如 Apache Hadoop，然后 MapReduce 从主数据集创建批处理视图。

实时处理层可以快速处理数据流，不需要完整的数据或 100% 的准确性。该层是弥补批处理层滞后的增量机制，可以给应用程序提供最新数据，这是数据质量与延时的一个折中方案。批处理完成后，实时处理层中的数据将替换为批处理层中的数据。该层通常由 Apache Storm 或 Spark 之类的流式技术实现，该技术的后端存储一般采用低延时数据存储，比如 Apache Cassandra。

最后，服务层将结果数据返回给应用程序，包括从批处理层创建的批处理视图，以及确保低延时查询的相关索引。该层通常采用 HBase、Cassandra、Apache Druid 或其他类似的数据存储。

除了实时处理层带来的低延时，该架构还有其他优势，尤其是主数据集的输入保持不变，这样在代码和业务规则更改时，可以重新处理数据。

该架构的最大缺点是，需要维护两个代码库，一个用于实时处理，另一个用于批处理。如果两个代码库不总是同步的，维护成本和数据完整性的风险将增加。更优秀的框架已经问世，可以将一份代码同时编译为实时处理层和批处理层；另外，操作和维护两个系统也很复杂。

Lambda 架构的另一个不足是，自该架构诞生以来，实时处理已非常成熟了。较新的流式处理系统可以在不牺牲延时的情况下保证批处理的语义。

2. Kappa 架构

Jay Kreps 在 LinkedIn 时首次阐述了 Kappa 架构的概念（见图 12-2）。在 Lambda 架构中，采用关系型数据库或 NoSQL 数据库持久化数据。在 Kappa 架构中，数据存储是只追加的不可变日志，例如 Kafka。实时处理层的流经过计算系统，并持久化到其他存储系统中。Kappa 架构消除了批处理系统，因为流式处理系统可以处理所有转换和计算。

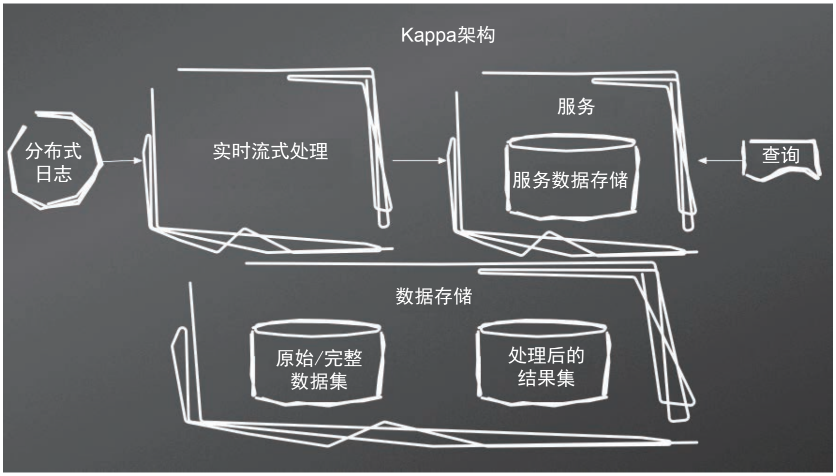


图 12-2：Kappa 架构

Kappa 的最大价值之一是，消除了批处理层，降低了 Lambda 架构的复杂性和运维成本，还缓解了迁移和数据重组的痛点。想要重新处理数据时，就可以重新处理、测试并切换。

Lambda 架构和 Kappa 架构是实时处理和呈现大数据的模式示例。下面介绍一些由数据驱动的架构模式，这些模式是应用程序直接使用数据存储这种传统方法的替代方法。

12.2.2 事件溯源

事件溯源是一种架构模式，颠覆了检索和插入数据的方式。数据存储抽象层的地位因此下降，从而为创建和重建数据视图提供了灵活性。

在事件溯源架构模式中，对实体的更改将保存为一系列状态的变更。当状态变更时，将新事件添加到日志中。在传统的数据存储中，变更是以当前状态替换之前的状态，因此具有破坏性。事件溯源架构模式中记录了所有变更，应用程序可以通过重放日志中的事件重建当前状态。这种数据存储称为**事件存储**。

事件溯源不仅是记录变更的日志，也是一种数据建模新模式。通过存储低级别的数据（事件而不是可以被覆盖的状态值），事件溯源扩充了传统存储，例如关系型数据库或键值数据库。

事件存储不仅充当事件的分布式日志和记录数据库，还充当消息系统。如前所述，下游流程和工作流可以订阅事件。当把事件保存在事件存储中时，该事件将转发给所有感兴趣的订阅者。可以采用关系型数据库、NoSQL 数据库或在分布式日志（例如 Kafka）中存储事件。甚至还有一个名为 EventStore（一个开源项目）的事件存储，用于存储仅追加的不可变记录。在很大程度上，选择取决于变更频率，以及快照和压缩之前事件存储所需的时长。

事件溯源有许多优点。与带来破坏性的变更不同，审计实体的生命周期将非常简单，调试和测试也变得更容易。采用事件存储，即使有人不小心删除了表或大块数据，也可以使用分布式日志根据丢失的特定实体，大规模地重建数据或恢复具体丢失的实体。但是，仍然存在挑战，尤其是管理 schema 的架构演变可能会影响存储的已有事件。在重放事件流时，外部依赖关系也可能很难重建。

因为事件溯源的诸多优点，即使在使用传统的数据存储而不是事件存储的情况下，很多应用程序依然实现了它。随着时间的推移，可以通过 API 提供完整的访问历史，进行审计、重建和不同的转换，好处巨大。

12.2.3 CQRS

从事件存储作为辅助存储，到作为核心数据存储层，是自然的演变，这就是命令查询责任分离（command-query-responsibility segregation，CQRS）。CQRS 的驱动力是使用多个模型或视图表示相同的数据。对不同模型的需求源自这样一种想法：不同领域有不同的目标，

并且这些目标需要不同的上下文、语言以及最终的数据视图。

这可以通过事件溯源实现。通过状态更改事件的分布式日志，订阅事件的工作进程可以构建有效的视图。CQRS 还具有其他一些有用的特性，不仅可以构建新视图，还可以针对不同的查询模式，通过不同的数据存储优化查询。举例来说，如果搜索的是文本数据，则将其存入像 Elasticsearch 之类的搜索存储中，为搜索应用程序创建优化的视图，还可以为每个聚合创建独立的扩缩容模式。通过针对查询进行了读优化的数据存储，以及针对写操作进行了优化的只追加日志，可以使用 CQRS 高效地分配和优化工作负载。

但是，该架构中可能存在不必要的复杂性。可以仅分离一个视图的数据，也可以过度分离视图。仅关注实际需要多模型方法的数据，对于降低复杂性很重要。

确保写操作或命令返回足够的数据，以有效地找到其模型的新版本，有助于长期降低应用程序的复杂性。如果命令返回成功 / 失败、错误以及获取结果模型的版本号，对此会有所帮助。甚至可以将模型的数据作为命令的一部分返回，这可能并不完全符合 CQRS 理论，但可以让大家的工作变得轻松。

无须将 CQRS 与事件溯源相结合。事件溯源作为核心数据存储机制非常复杂，应确保这种方式仅用于表示有一系列状态更改的数据。可以使用视图、数据库标志、外部逻辑，或其他任何比事件溯源更简单的方式实现 CQRS。

这只是数据驱动架构的示例，你可能正在使用或设计此类架构。关键是明确数据的生命周期，并找到有效的存储和传输方式，以便将数据发送到系统的各个组件。可以以多种方式表示数据。当今大多数组织最终需要实现多种表示模型，同时保持核心数据集的完整性。

12.3 小结

本章介绍了在数据存储中增强功能、可用性、扩展性和性能的方法。所述复杂架构有很多应用场景，但这些架构的维护成本很高。当然，最糟糕的情况是数据的完整性受损。在你的整个职业生涯中，这是很严肃的问题。

第 13 章将讨论如何继续发展自己的事业，并针对如何在组织内部创建数据库可靠性文化提供一些指导意见。

第13章

数据库可靠性工程师行为指南

数据库工程领域这些年的变化是贯穿本书的话题。在此背景下，我们讨论了数据库可靠性工程师必须参与的运维和开发工作，以及如何开展这些工作。下面探讨当前存储、复制、数据存储和架构的生态系统（至少是一个合理的子集），以拓展思维和知识。

如头衔所示，针对数据库可靠性工程师所做的所有事情都需要强调可靠性，因为数据库不得存在风险和混乱。如今，在数据库可靠性工程师的日常工作中司空见惯的技术——虚拟化、基础设施即代码、容器、无服务器计算和分布式系统——都源自可以容忍风险的计算领域。现在这些基础设施无处不在，因而作为组织最宝贵资源（数据）的管理者，数据库可靠性工作师需要找到将数据库融入这些范式的途径。

这份工作很崇高。当谈论数据时，任何组织内部都只能容忍极低的风险。因此，如何将这些概念引入组织的其他部门，或者在其他人这样做时如何回应，是数据库可靠性工程师的工作准则和职责。拥有愿景和意志还不够，必须找到实现愿景的方法。

本章讨论如何在组织内培养数据库可靠性文化，以及如何立足于数据库可靠性工程参与组织内部的各种事务。

13.1　数据库可靠性工程文化

数据库可靠性工程的文化是怎样的？如何推广？可靠性工程文化通常具有以下共性：

- 就事论事地进行反思；
- 将重复性工作自动化；
- 结构化与合理的决策。

这一切都是有意义的，运维团队或网站可靠性工程师团队的每个人都应为此不断努力。但是，作为（或想成为）数据库可靠性工程师，应该怎么做呢？下面探讨如何引入可靠性文化，以及如何将数据库可靠性工程师的专业知识传播到组织的其他部门。

13.1.1　突破障碍

与其他团队（有数据存储需求的团队）联系不密切的数据库管理员，根本不会成功。为了积极地发挥作用，相比传统角色，我们需要在更高的抽象层次上成为活跃的团队成员和合作伙伴。这样做存在固有的挑战，因为数据库相关职位的人员通常不多。本书一直强调，数据库相关人员根本无法扩展到与开发人员和运维工程师相当的数量。

在某些方面，数据库可靠性工程师可以证明自己是跨职能部门的高效能员工。数据库可靠性工程师参与跨职能工作时应该怀抱目标——致力于提供专业知识并消除资源方面的限制，或者更多地了解其他职能，以提高在组织中的运作能力。

1. 架构流程

毋庸置疑，那些拥有深厚的数据库专业知识的人，应该更多地参与架构过程的所有阶段，尤其是设计阶段。在选择数据存储时，数据库可靠性工程师可以提供有价值的数据，尤其是已经在生产中经过大量测试和检验的数据存储。正如第 7 章和第 11 章所述，数据库可靠性工程师负责审查组织将使用的数据存储。

在需要自助服务来构建和部署服务的大型组织中，数据库可靠性工程师有权决定哪些存储服务采用自助服务。通过与其他技术部门合作，数据库可靠性工程师可以提供经过批准的服务，这些服务已经针对边缘情况、扩展性、可靠性和数据完整性进行了全面测试。有时，除了已批准的服务，所有部门都可以构建和部署服务，但这样做必须接受不同的 SLA，示例如下。

- **一级存储**

  在生产环境中对核心服务进行了测试。采用自助服务模式，意味着运维人员和数据库可靠性工程师能为升级提供 15 分钟的 Sev1 响应 SLA，并在可用性、延时、吞吐量和持久性方面保证最高的 SLA。

- **二级存储**

  在生产环境中对非关键服务进行了测试。采用自助服务模式，意味着运维人员和数据库可靠性工程师可以提供 30 分钟的 Sev1 响应 SLA，并在可用性、延时、吞吐量和持久性方面保证一定的 SLO。

- **三级存储**

  未经生产测试。对运维人员和数据库可靠性工程师而言是尽力而为的，不保证 SLO，必须得到软件工程团队的全力支持。

如果组织不支持此类自助服务平台，则数据库可靠性工程师必须更加努力，确保每个团队都知道在评估数据存储和架构时所用的方法以及这样做所带来的价值。尽管过早优化是有风险的，但数据库可靠性工程师的职责之一，就是确保数据存储的架构决策在将来达到系统的扩展拐点时不会阻碍服务。

在许多组织中，这是从技术项目的必要清单开始的，该清单包括在投入生产之前审查所有项目的数据库。但是，这样做很容易造成在项目的生命周期中介入较晚，甚至超出了更改范围，这时就需要制度发挥作用了。花时间评估组织中普遍使用或即将采用的数据存储并发布最佳实践以及其中的权衡和范式，是展示数据库可靠性工程师价值的重要方式，也是与软件工程师和架构师合作构建数据存储及共享知识的绝佳机会。随着定期发布，当架构中引入尚未经过评估的新数据存储时，鼓励大家一起评估新数据存储作为项目和资源一部分的优缺点。关键是让大家看到数据库可靠性工程师的价值，而不是限制。

可以使用一些度量值来衡量数据库可靠性工程师和组织在这方面的表现，示例如下。

- 多少项目的架构有数据库可靠性工程师参与或采用了其批准的模板？（事后回顾）
- 使用和部署了什么存储？（事后回顾）
- 对于每个项目阶段或者用户故事[1]，数据库可靠性工程师做了多少工作？（事后回顾）
- 度量存储层和引擎的可用性、吞吐量和延时，以此衡量可靠性。

2. 数据库开发

与构建架构类似，数据库可靠性工程师尽早参与数据库项目的开发，有助于项目成功。第 8 章讨论过这类实践及其价值。这方面的最大障碍之一是，软件工程师忘记与数据库可靠性工程师讨论他们的设计；有时软件工程师认为不需要这种指导。这种“斗争”有助于软件工程师团队看到与数据库可靠性工程师合作所带来的价值。

数据库可靠性工程师应与开发团队一起工作，无论全职参与还是仅参与一段时间。和软件工程师结对工作，有助于让开发团队看到与数据库可靠性工程师协作的价值。即使数据库可靠性工程师不擅长编写代码，他们对数据建模、数据库访问和功能方面的建议也是很有价值的，也有助于增进团队之间的关系。软件工程师与数据库可靠性工程师（负责审阅、实现或值班以支持数据驱动型应用）结对工作，可以促进团队之间的关系、相互理解以及跨团队的知识传播，进而加快开发进度。

前面也讨论了在数据层的集成方面，为软件工程师提供最佳实践和模式的重要性。例如，可以为模型、查询或者功能列清单，并请软件工程师反馈是否采用了该模式。这些清单可以标记出需要审阅后才能发布到生产中的用户故事或功能。

有一些指标可以衡量数据库可靠性工程师和组织在这方面的表现，示例如下：

注 1：敏捷开发中的概念。——译者注

- 软件工程师与数据库可靠性工程师结对工作的时间；
- 数据库可靠性工程师参与的用户故事数；
- 将功能特性的度量值（比如延时和持久性）与是否有数据库可靠性工程师参与相对应；
- 与软件工程师结对轮值。

3. 生产环境变更

毋庸置疑，所有人都希望无差错地执行变更，但数据库可靠性工程师无法应对快速攀升的部署频率。最后，积压的待办事项被揉进一个庞大、脆弱的变更集中，这可能会带来巨大的风险；或者，在未经数据库可靠性工程师审阅的情况下就执行了变更。如第 8 章所述，应对这种情况的有效方式是构建流程和工具，以便在部署时软件工程师能够更好地决策：哪些可以通过常规部署机制完成，哪些需要数据库可靠性工程师执行，以及在不确定的情况下哪些内容需要数据库可靠性工程师审阅。

第一步最为简单——逐步建立一个探索性工具库，它可以指出一项变更是否安全。即使无法立即消除瓶颈，但和数据库可靠性工程师一起创建该工具库将会逐渐发挥作用。然后，成立一个评审委员会，定期总结近期的变更、探索性尝试和指导的效果，以及这些变更是否成功。该委员会将有效监督该流程。可以将其视作定期回顾变更的一部分，无论成功与否。

促进软件工程师团队尽可能自主的另一种方式是，建立一个数据库的变更模式，探索性地将其应用于将来的变更中。慢慢地，当与软件工程师结对工作并按该方法处理变更时，就可以新建一个动态文档，软件工程师可以使用该文档并逐步完善。此外，回顾和检查该模式是否成功至关重要。

可以通过为探索性尝试和变更模式提供保护措施，来进一步完善软件工程师所使用的框架。这些保护措施能让所有人（软件工程师、运维人员、数据库可靠性工程师和领导层）更有信心。这种机制运转得越好，大家的信心越强，效果就越好。通过让软件工程师团队自助完成，你与这些团队的关系会变得更加融洽，因为你证明了自己的价值。继续发挥你在数据库存储和访问方面的专长，将会进一步加快他们的开发速度，你们的关系也会更融洽。可以采用结对开发或培训的方式（比如研讨会、知识分享和文档）。

一些变更必须有数据库可靠性工程师参与，大量赋能也不能令其动摇。即使在该阶段，仍然可以采取培训或为他人赋能的方法。同样，在计划和执行变更时与工程师结对是很好的做法。和运维工程师结对也很有价值，因为协助处理复杂生产环境的人员越多越好。

即使缺少有效的自动化，数据库可靠性工程师和团队仍有很多方法来实施稳定、无差错的变更，而不会导致开发流程停滞。在完善手工流程的可靠性和可复用性之后，技术、工具和代码可以提供进一步的保证。

有些度量值可以衡量数据库可靠性工程师和组织在这方面的表现，示例如下：

- 软件工程师 / 运维人员在变更中与数据库可靠性工程师结对工作的时间；

- 需要数据库可靠性工程师参与的变更和全部变更的数量对比；
- 变更失败的次数及其影响。

4. 基础设施设计与部署

之前讨论过和工程师一起选择经过测试、可靠的数据存储。同样，你必须和运维人员以及基础设施人员协作，确保相关必要事项都已准备就绪，不仅包括托管数据存储的资源，还包括部署和维护所需要的一切。第 5 章详细讨论了上述事项的各个方面，第 6 章介绍了管理大规模基础设施所需的软件和工具。但对于数据存储（尤其是分布式数据存储），仍有很多事要做。

与生产环境实施变更并给予软件工程师更多自主权相似，将其引入组织旨在逐渐建立信任。给数据库可靠性工程师团队和组织带来显著价值的第一步，是采用相同的代码仓库和版本管理系统管理脚本、配置文件以及文档。然后，可以和运维团队一起通过配置管理和编排，配置并部署空数据存储。最终，仍然需要将实际数据写入数据存储中，只不过是逐步推进的。

在整个过程中，通过和运维人员结对，可以进行配置测试、安全性测试、负载测试，甚至是数据完整性测试和备份测试这样的高级测试。整个团队越熟悉数据库的工作原理以及故障的起因，效果就越好。可用性测试和故障测试也是需要与其他团队合作完成的重要测试。

最后，可以让运维人员，甚至管理自己基础设施的高级工程师作为主要值班人员。通过结对工作的方式，他们能从使用这些基础设施中迅速建立信心，同时最大限度地降低风险。只有当团队真正了解如何维护这些基础设施的各个方面时，你才能着手将风险更高的组件自动化，例如数据加载、复制修复和主节点故障转移。

有些度量值可以衡量数据库可靠性工程师和组织在这方面的表现，示例如下：

- 使用配置管理系统的基础设施组件的数量；
- 集成到编排平台的基础设施组件的数量；
- 部署成功或失败的计数；
- 资源消耗度量值——数据存储使用的所有子系统；
- 非数据库可靠性工程师值班的次数；
- 由非数据库可靠性工程师处理的故障和 MTTR；
- 故障升级到需由数据库可靠性工程师处理的次数。

这项工作是否成功，主要取决于关系、理解、信任和知识共享。许多数据库管理员习惯于单打独斗，而通过这些步骤可以将数据库相关工作引向光明大道。在数据领域，不再是只有最勇敢或最愚蠢的工程师才愿意解决模糊的难题。关键是反复接触、不断建立信任，以及与他人结对。

13.1.2 数据驱动决策

在没有相关数据（关于变更之后的影响）支撑的情况下，无法建立信任。戴明环中关于计

划、执行、检查和行动的要求，第 4 章已有论述。每次变更前，要明确定义成功的指标，并以质疑的态度分析变更结果。

如第 2 章和第 3 章所述，利用组织 SLO 的相关知识，对于理解变更的必要性、度量值和结果（向组织其他部分证明潜在价值以推动变更）以及努力的价值至关重要。

希望你所在的组织了解数据驱动决策的价值，并且实施了观测平台、流程分析以及行为准则，而且定义了清晰、有用的 SLO，以推动决策。若非如此，则需要从这些实践开始，以便带来更深远、更广泛的影响。

13.1.3　数据完整性和可恢复性

第 7 章讨论了数据完整性的重要性，以及在数据丢失或损坏的情况下如何恢复。组织通常将其视为数据库可靠性工程师的责任，然而数据库可靠性工程师团队不可能独自完成这项任务。作为数据完整性的拥护者，数据库可靠性工程组织通常要承担责任。说服软件工程师团队为数据验证流水线和恢复 API 分配更多资源，是一项持续性的职责。如果解决了架构和软件开发早期缺少数据库可靠性工程师参与的困难，就会增强人际关系和信任，进而逐步构建共享代码，这有利于实现数据完整性验证流水线。

这并非易事，我们的经验是大多数软件工程师认为数据完整性仅是数据库可靠性工程师要考虑的事情。资源匮乏的组织不愿开发验证流水线和用于恢复的 API，因此，必须采用“草根”思维的解决方案，与数据完整性问题周旋。同样，追踪手动恢复数据所付出的人力成本，对于说服领导为开发用于恢复的 API 和验证流水线投入资源大有帮助。

如前所述，数据库可靠性的成功演进要求组织逐步、全面地转变。选择占用你很多时间且对其他团队造成极大限制的领域，是一项需要明智地实践的技能。然后，通过组织观念的不断转变来逐步建立信任并改进工作方式，保持前进的势头。而这一切都需要时间、信任和大量实验，才能确定哪些对组织所处的风险等级是有效的，哪些是无效的。

13.2　小结

感谢抽出宝贵时间阅读本书。我们对发展这种复杂的、具有挑战性的技术职业充满热情。尽管书中相当一部分内容比较新颖或者仍在验证中，但我们认为数据库可靠性工程师的观念转变可以为数据驱动型服务和组织带来可观价值。

希望你受到启发去探索组织中的这些变化，并渴望学习更多知识。由于该框架比较灵活，因此我们提供了可供参考和深入研究的阅读资料。但是，更重要的是展示传统的数据库管理员如何融入现代和未来世界。数据库管理员的角色不会消失，无论你是行业新人还是资深人士，都愿你拥有长远的职业生涯，为你所在的组织贡献价值。

关于作者

莱恩·坎贝尔（Laine Campbell）在 Fastly 担任生产工程高级总监，她还是 PalominoDB/Blackbird 的创始人和前首席执行官。PalominoDB/Blackbird 为众多公司提供数据库需求咨询服务，其客户包括 Obama for America、Activision Call of Duty、Adobe Echosign、Technorati、Livejournal 和 Zendesk。她在运行大型数据库和分布式系统方面有超过 18 年的工作经验。

夏丽蒂·梅杰斯（Charity Majors）是 honeycomb.io 的首席执行官和联合创始人。honeycomb.io 结合了日志聚合器的原始准确性、时序指标的速度和 APM（应用性能指标）的灵活性，提供了首个真正意义上的下一代分析服务。她曾在 Parse 和 Facebook 负责运营工作，管理大量的 MongoDB 复本集以及 Redis、Cassandra 和 MySQL。她还与 Facebook 的 RocksDB 团队密切合作，使用可插拔存储引擎 API 开发出了世界上首个使用 RocksDB 作为底层存储引擎的 MongoDB。

封面介绍

本书封面上的动物是萨福克马，也称萨福克矮马或萨福克栗毛马。这种原产自英国的挽马一般通体栗色，腿部粗壮有力。

萨福克马出现于 16 世纪，用于农场工作。虽然该品种在 20 世纪初受到欢迎，但由于农业机械化的发展，到 20 世纪中叶数量大幅减少。萨福克马高 165~178 厘米，重 900 多千克。它们体型较矮，但仍比其他英国挽马（比如克莱兹代尔马和夏尔马）庞大。

O’Reilly 图书封面上的许多动物濒临灭绝，它们是这个世界所剩无几的瑰宝。如果想了解如何为这些动物提供帮助，请访问 animals.oreilly.com。

封面图片来自大英博物馆。

O'REILLY®

数据库可靠性工程：数据库系统设计与运维指南

信息技术领域的基础设施即代码革命正在影响着数据库管理。本书从实践角度介绍如何将网站可靠性工程的实践应用于数据库架构和运维，为想成为数据库可靠性工程师的专业人士提供一个指导框架，开发人员、系统管理员和初中级数据库管理员等都能从中受益。

本书首先介绍数据库可靠性工程师需要掌握的核心运维概念，然后介绍各种数据库持久性选项，包括如何利用关键技术提供有弹性、可扩展、高性能的数据存储和检索。有了数据库可靠性工程的坚实基础，读者便能深入研究任何现代数据库的架构和运维。

- 服务等级要求和风险管理
- 构建并演进具有运维可见性的架构
- 基础设施工程和基础设施管理
- 如何推动发布管理流程
- 数据存储、索引和复制
- 明确数据存储的特性和用例
- 数据存储架构组件和数据驱动的架构

“这本书从实践经验和重要细节出发，探讨了网站可靠性工程师解决重要问题的视角和措施。建议任何负责关键业务数据的软件工程师都读一读这部领域标杆之作。”

——**John Allspaw**
Etsy前CEO

莱恩·坎贝尔（Laine Campbell）是Facebook生产工程总监，曾任Fastly生产工程高级总监，PalominoDB/Blackbird创始人、前CEO，为众多公司提供数据库需求咨询服务，在运行大型数据库和分布式系统方面有近20年经验。

夏丽蒂·梅杰斯（Charity Majors）是Honeycomb公司联合创始人和CTO，曾任Facebook生产工程经理，是一位有15年以上经验的高级系统工程师。

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn

分类建议　计算机/数据库

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

扫码加入
图灵读者群

ISBN 978-7-115-54886-3

定价：89.00元